

RAZOR COMPONENTS

SUCCINCTLY

BY ED FREITAS

Razor Components Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2021 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

ISBN: 978-1-64200-211-9

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Graham High, senior content producer, Syncfusion, Inc.

Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the <i>Succinctly</i> Series of Books	7
About the Author	9
Razor Components	11
Chapter 1 Setup and Fundamentals	12
Overview	12
Defining a component	12
Bootstrapping the app	12
Adding the Skclusive-UI library	16
Changing the NavBar color	16
Adding Skclusive-UI references	17
Updating App.razor	19
Updating the _Imports.razor file	20
Replacing Bootstrap with Material Design for Bootstrap	22
Hot reload	22
Running the app	24
Summary	25
Chapter 2 Component Fundamentals	26
Component basics	26
Component reusability	28
Creating a new component	30
How components are rendered	33
Component differences between models	35
Page routing and rendering	35
Component code organization	39

Hidden code-behind	41
Component inheritance	43
Razor file nesting.....	45
Partial classes	46
Switching back	49
Summary.....	51
Chapter 3 Component Features	52
Event handling	52
One-way data binding.....	55
Child content	56
Class libraries and component sharing.....	58
Summary.....	62
Chapter 4 Using Components.....	63
Conditional rendering	63
Invoking ForecastRow.....	67
Object injection and initialization.....	70
Extras object model.....	71
Extras service.....	71
Registering the service	73
Updating ForecastRow.razor	73
Life cycle methods.....	75
Element preservation.....	76
Two-way data binding.....	77
Event callbacks	81
Combining parameters	87
Referencing components.....	93

Summary.....	97
Chapter 5 Templating	98
The case for component templating.....	98
Table template.....	98
Closing comments.....	102

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge
As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas is a consultant on software development related to financial process automation, accounts payable processing, and data extraction.

He loves technology and enjoys playing soccer, running, traveling, life hacking, learning, and spending time with his family.

You can reach him at <https://edfreitas.me>.

Acknowledgments

Many thanks to all the people who contributed to this book, including the amazing [Syncfusion](#) team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Jacqueline Bieringer from Syncfusion and [James McCaffrey](#) from [Microsoft Research](#). Thank you.

This book is dedicated to *Puntico*—may your journey be blessed—and to my father. For everything you did, for everyone you loved—thank you.

Razor Components

[Blazor](#), which started as an experimental framework developed by [Microsoft](#), is a [single-page application](#) development framework whose key feature is to enable developers to write entire web applications in [C#](#) and execute them in the browser. To achieve this, a .NET runtime that works with [WebAssembly](#) is used.

As part of Blazor's evolution, a server-side model was introduced where the C# code runs on web servers, using a [SignalR](#) connection to make updates to the application in the browser in real time. This model was officially integrated as part of .NET Core 3.0 in Preview 2 and was named, at the time, Razor components.

Razor components form part of the Blazor framework, and the term now refers to specific building blocks within the Blazor framework itself rather than the server-side model of the development framework, which is now referred to as server-side Blazor or Blazor Server apps.

Within the Blazor framework, Razor components are created in files with the **.razor** extension and they can perform many roles: representing a specific piece of the user interface, a view component, or a tag helper.

Razor components can also represent a layout or an entire page. When referenced within a Razor page or an MVC view, they will take the place of a partial, a view component, or a custom tag helper.

Blazor is an excellent framework for writing single-page applications while using just C# as the programming language, and for that, the *Succinctly* series has you covered with [Blazor Succinctly](#) and [Blazor WebAssembly Succinctly](#).

In this book, we will explore how to create and work with both simple and advanced Razor components. This book will not cover how to create a full Blazor application.

First, we will explore how to write a basic component using one-way data binding and events. After that, we'll have a look at two-way data binding, event callbacks, life cycle methods, and component references.

Finally, we will explore how to enable component reuse by creating a component template. After reading this book, you will have a good understanding of the skills required to create amazing Razor components for any Blazor application.

I invite you to join me on this fascinating journey to learn the ins and outs of creating Razor components and to explore code reusability using the amazing Blazor framework.

Chapter 1 Setup and Fundamentals

Overview

We are about to embark on a captivating journey and are almost ready to begin exploring Razor components.

Before we can jump into Razor components though, we'll need to bootstrap a Blazor Server application, which is what this chapter is all about.

To make the most of the concepts that will be covered throughout this book, I recommend that you explore [Blazor Succinctly](#) by Michael Washington, which is an amazing resource to get you up to speed with Blazor development—as we won't cover the fundamentals of creating full-blown Blazor applications, but instead focus mainly on Razor components.

The concepts that we will cover throughout this book are applicable for both Blazor Server applications as well as Blazor WebAssembly apps. Sounds exciting, so let's get started.

The full source code and Visual Studio solution for the application that will be developed through the course of this book can be downloaded from this [URL](#).

Defining a component

A Razor component is nothing more than a reusable piece of the user interface in a Blazor application.

Blazor applications are made of multiple components. Some of these components are organized into Blazor pages; however, some of them might be reusable components that can be shared among multiple pages.

Bootstrapping the app

Throughout this book, we will use and add components to a Blazor Server application, which we will bootstrap using the excellent [Skclusive-UI](#) library. The code for the bootstrapped app, which we will use as our starting point, can be found [here](#).

To get started, make sure you have [Visual Studio 2019](#) installed. Once Visual Studio is up and running, go to **File > New > Project**, and type **Blazor** in the search box. There, choose the **Blazor App** template.

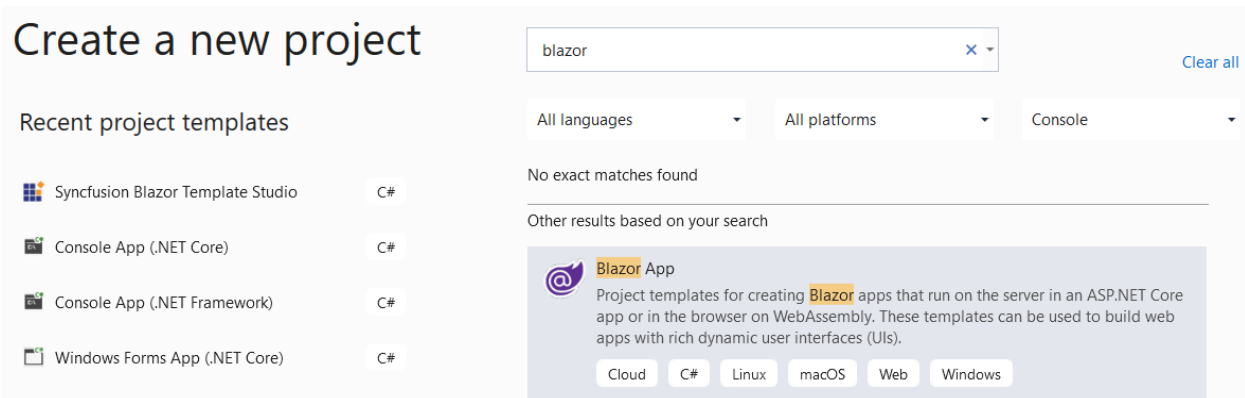


Figure 1-a: The Blazor App Template

Once you have selected the **Blazor App** template and clicked **Next**, you'll be presented with a screen where you can change the name of the project and the directory where it resides.

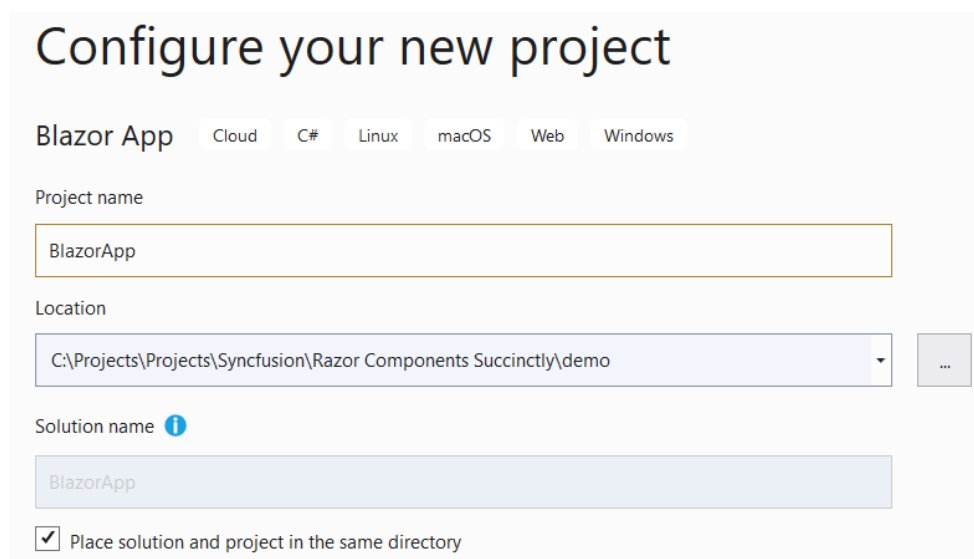


Figure 1-b: Configuring the Blazor Project Name and Folder

You can leave the default project name (**BlazorApp1**) as it is, but for the example code in this book I renamed the project to just **BlazorApp**. In a real-world scenario, I would recommend that you choose a more descriptive project name. Once you have chosen the project folder (**Location**), you can create the project.

Once you have clicked the **Create** button, you'll be shown the following screen. There, you will be able to choose the **Project Type**.

Create a new Blazor app

Blazor Server App
A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication
No Authentication
[Change](#)

Advanced
☒ Configure for HTTPS
☐ Enable Docker Support
(Requires [Docker Desktop](#))
Linux

Author: Microsoft
Source: Templates 3.1.6

[Get additional project templates](#)

Back Create

Figure 1-c: Selecting the Blazor Server App Project Type

In my case, I'll be choosing the **Blazor Server App** template. You are also free to choose **Blazor WebAssembly App**.

Nevertheless, if you want to follow along easily with the steps that will be described in this book, I suggest you choose the **Blazor Server App** template, as there are some slight differences between the templates when making changes to the bootstrapped application.

The Razor components concepts that will be described in this book apply to both Blazor Server and Blazor WebAssembly apps.

Once you have selected the template, click the **Create** button to finalize the creation of the application.

Once the application has been created, you will be able to see it within **Solution Explorer** in Visual Studio, which we can see as follows.

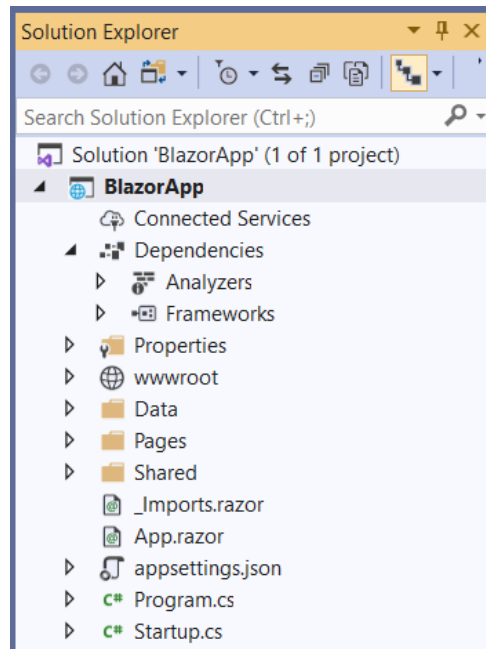


Figure 1-d: The Application within Solution Explorer (Visual Studio)

Next, we can execute the application by clicking the button highlighted in the following figure.

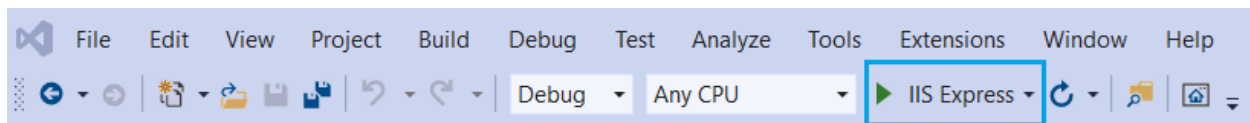


Figure 1-e: The Execute App Button in Visual Studio

Once the application executes, a browser window will open and we'll see the following screen.

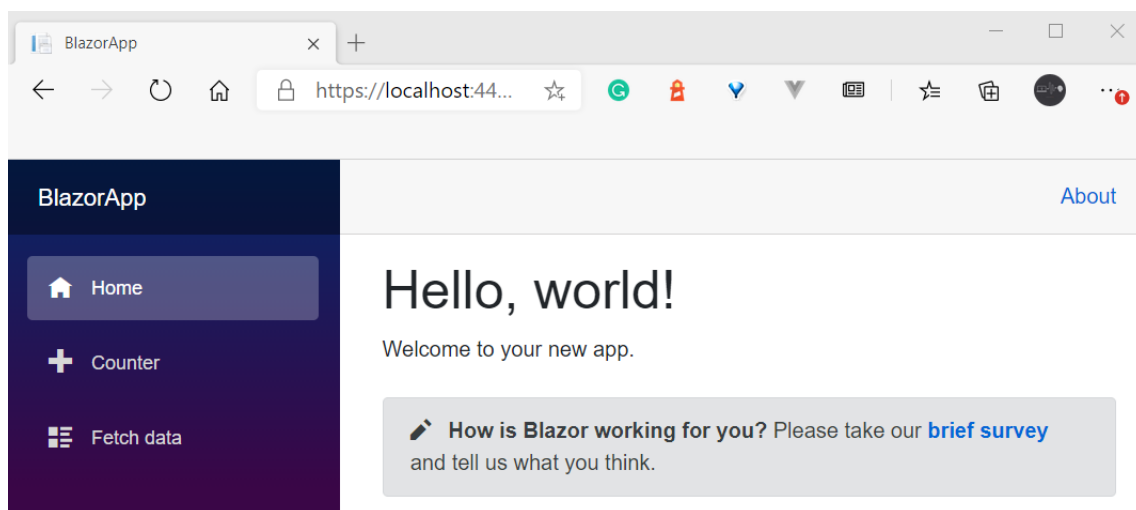


Figure 1-f: The Application Running

Awesome—we have successfully scaffolded a Blazor Server application. Now, let's add the Skclusive-UI library.

Adding the Skclusive-UI library

To add the Skclusive-UI library, within Visual Studio, click on the **Tools** menu, go to the **Command Line** option, and then click the **Developer Command Prompt** item. This will open a command prompt.

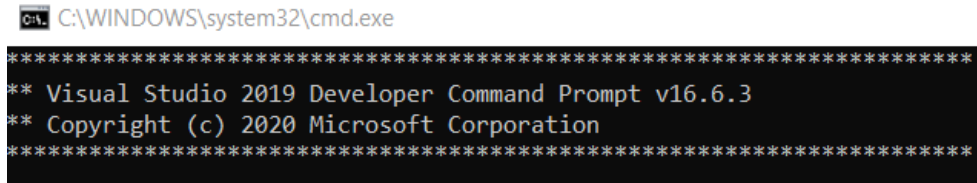


Figure 1-g: The Developer Command Prompt

From the **Developer Command Prompt**, run the following command.

Code Listing 1-a: Adding the Skclusive-UI Library

```
dotnet add package Skclusive.Material.Component
```

After this command has been executed, the Skclusive-UI library will be installed. Before we start to dive into the fundamentals of Razor components, let's make some changes to the Blazor application we've just created.

These changes include modifying the **NavBar** color, adding all the references to the Skclusive-UI library, replacing [Bootstrap](#) (which comes out of the box with Blazor) with [Material Design for Bootstrap](#), and also adding hot reload (which doesn't come out of the box with Visual Studio 2019) for Blazor projects.

Changing the NavBar color

In Visual Studio, open **Solution Explorer**, and within the **wwwroot** folder, under the **css** subfolder, open the **site.css** file.

Within the **site.css** file, locate the **sidebar** CSS class and remove the **background-image** line, which is highlighted in bold in the following listing.

Code Listing 1-b: The sidebar CSS Class (Before)

```
.sidebar {  
    background-image: linear-  
gradient(180deg, rgb(5, 39, 103) 0%, #3a0647 70%);
```



```
}
```

Replace the line that was removed with the following **background-color** line (highlighted in bold).

Code Listing 1-c: The sidebar CSS Class (After)

```
.sidebar {  
    background-color: #0061f4;  
}
```

By doing this, our application's **NavBar** will have a color that will be more aligned with the [Material Design](#) look and feel.

Adding Skclusive-UI references

With Visual Studio open, go to **Solution Explorer** and open the **Startup.cs** file, which can be found in the project's root folder.

Within the **ConfigureServices** method, we need to add a new service—highlighted in bold in the following code listing.

Code Listing 1-d: ConfigureServices Method (Startup.cs)

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddRazorPages();  
    services.AddServerSideBlazor();  
    services.AddSingleton<WeatherForecastService>();  
  
    // New service  
    services.TryAddMaterialServices(new MaterialConfigBuilder().Build());  
}
```

For this to work, we need to add the **using** statement highlighted in bold in the following listing to the **Startup.cs** file.

Code Listing 1-e: Using Statements (Startup.cs)

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Builder;
```

```

using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using BlazorApp.Data;

using Skclusive.Material.Component; // New

```

The complete code for the updated **Startup.cs** file is shown in the next code listing. Notice that the namespace name is the same as the project name, so if you used a different project name when you created the application, your namespace name will not be **BlazorApp**.

Code Listing 1-f: Updated Startup.cs File

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using BlazorApp.Data;

using Skclusive.Material.Component; // New

namespace BlazorApp
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {

```

```

        services.AddRazorPages();
        services.AddServerSideBlazor();
        services.AddSingleton<WeatherForecastService>();
        services.TryAddMaterialServices(
            new MaterialConfigBuilder().Build()); // New
    }

    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapBlazorHub();
            endpoints.MapFallbackToPage("/_Host");
        });
    }
}

```

Updating App.razor

Next, we need to update the **App.razor** file, which can be found within the project's root folder. So, to do that, go to **Solution Explorer** and open the **App.razor** file.

Add the following lines before the **Router** component, highlighted in bold as follows.

Code Listing 1-g: Updated App.razor File

```
<!--New-->
<MaterialScripts />
<MaterialStyles />

<!--New-->
<ThemeProvider Theme="@Theme.Light" />

<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData"
      DefaultLayout="@typeof(MainLayout)" />
  </Found>
  <NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
      <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
  </NotFound>
</Router>
```

Notice that we are configuring the app to use the light theme of the Skclusive library, which is done by setting the **Theme** property of the **ThemeProvider** component to **@Theme.Light**.

Updating the _Imports.razor file

To be able to make the components from the **Skclusive** library available to all the pages of our Blazor application, we need to add the required references to use the **Skclusive** library.

To do this, go to **Solution Explorer** and open the **_Imports.razor** file, which can be found in the project's root folder.

There, add the following statements that reference the **Skclusive** library (highlighted in bold).

Code Listing 1-h: Updated App.razor File

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop

@using BlazorApp
```

```
@using BlazorApp.Shared

// New
@using Skclusive.Core.Component
@using Skclusive.Transition.Component;
@using Skclusive.Script.DomHelpers

@using Skclusive.Material.Core
@using Skclusive.Material.Component
@using Skclusive.Material.Theme
@using Skclusive.Material.Form
@using Skclusive.Material.Input
@using Skclusive.Material.Text
@using Skclusive.Material.Typography
@using Skclusive.Material.Progress
@using Skclusive.Material.Transition
@using Skclusive.Material.Button
@using Skclusive.Material.Icon
@using Skclusive.Material.Paper
@using Skclusive.Material.Avatar
@using Skclusive.Material.Badge
@using Skclusive.Material.Selection
@using Skclusive.Material.Toolbar
@using Skclusive.Material.AppBar
@using Skclusive.Material.Divider
@using Skclusive.Material.Grid
@using Skclusive.Material.Table
@using Skclusive.Material.Baseline
@using Skclusive.Material.Card
@using Skclusive.Material.Hidden
@using Skclusive.Material.List
@using Skclusive.Material.Modal
@using Skclusive.Material.Script
@using Skclusive.Material.Drawer
@using Skclusive.Material.Dialog
@using Skclusive.Material.Popover
@using Skclusive.Material.Menu
@using Skclusive.Material.Tab
@using Skclusive.Material.Container
```

Replacing Bootstrap with Material Design for Bootstrap

Next, let's replace Bootstrap with Material Design for Bootstrap. This can be done by going to **Solution Explorer**, navigating to the **Pages** folder, and opening the **_Host.cshtml** file.

With the file open, comment out the references to the Bootstrap and Font Awesome libraries, which are highlighted in bold and green in the next code listing, add a new reference to Font Awesome that includes Material Design icons, and add a reference to the Material Design for Bootstrap library hosted on unpkg.com.

The modified **head** section of the **_Host.cshtml** file is shown in the following Code Listing.

Code Listing 1-i: Updated Head Section of _Host.cshtml

```
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
  initial-scale=1.0" />
  <title>BlazorApp</title>
  <base href="~/ " />
  <!--
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto
:300,400,500,700&display=swap" />-->
  <!--<link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />-->

  <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Ro
boto:300,400,500,700|Material+Icons">
  <link rel="stylesheet" href="https://unpkg.com/bootstrap-material-
design@4.1.1/dist/css/bootstrap-material-design.min.css" integrity="sha384-
wXznGJNEXNG1NFsbm0ugrLFMQPWSwR3lds2VeinahP8N0zJw9VWSopbjv2x7WCvX" crossorigin
in="anonymous">

  <link href="css/site.css" rel="stylesheet" />
</head>
```

Hot reload

When creating the UI of a Blazor Server application, it is useful to see the changes without having to rebuild the project every time.

This is known as hot reloading, and it is a feature that is widely used when building [React](#), [Vue](#), or [Flutter](#) applications.

Ideally, the browser should be able to refresh automatically after making a change to the UI. However, that is something that is not yet available in Visual Studio 2019 at the time of writing of this book.

Fortunately, through a workaround, it is possible to implement hot reloading using a public Blazor API, which can trigger a browser reload when it loses a [SignalR](#) connection.

So, let's add this capability to our project. To do so, go to **Solution Explorer** and open the **_Host.cshtml** file, which resides in the **Pages** folder.

Just before the **</body>** tag, add the following code.

Code Listing 1-j: Adding Hot Reload (**_Host.cshtml**)

```
<!--New-->
<environment include="Development">
  <script src="~/Scripts/HotReload.js"></script>
</environment>
```

If you read carefully, you'll notice that the **HotReload.js** file doesn't exist within the project folder structure and neither does the **Scripts** folder.

Going back to **Solution Explorer**, navigate to the **wwwroot** folder, and within it create a new folder called **Scripts**.

In this folder, let's create a new JavaScript file called **HotReload.js**. This can be done by right-clicking on the **Scripts** folder within **Solution Explorer**, and then choosing the **Add > New Item** menu option.

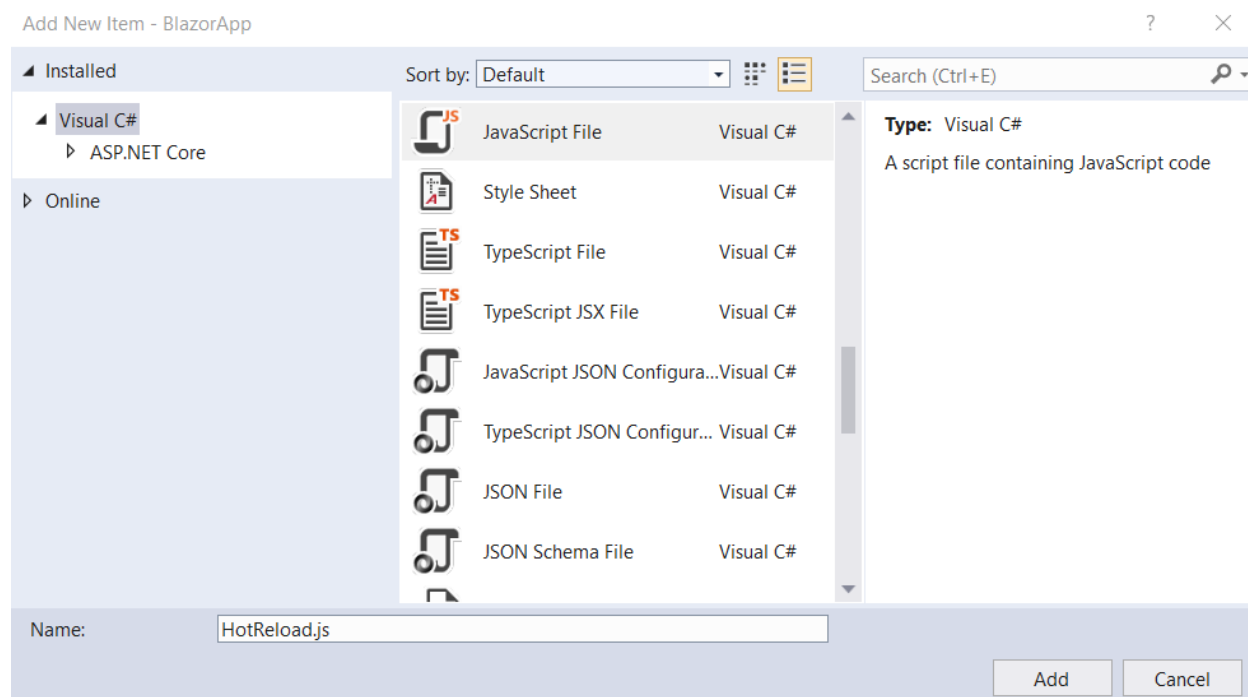


Figure 1-h: Adding **HotReload.js**

To create the **HotReload.js** file, click **Add**. Once the file has been created, open it and add the following code.

Code Listing 1-k: HotReload.js

```
window.Blazor.defaultReconnectionHandler.onConnectionDown = function () {  
    window.location.reload();  
};
```

As you can see, what the preceding code does is bind an anonymous function to the **onConnectionDown** event. This executes the **window.location.reload** method, which forces the page to reload after the UI changes.

Running the app

Now that we have modified the application, let's execute it and test the hot reload functionality. To do that, go to Visual Studio and press **Ctrl+F5**. This will open a browser window, which in my case looks as follows.

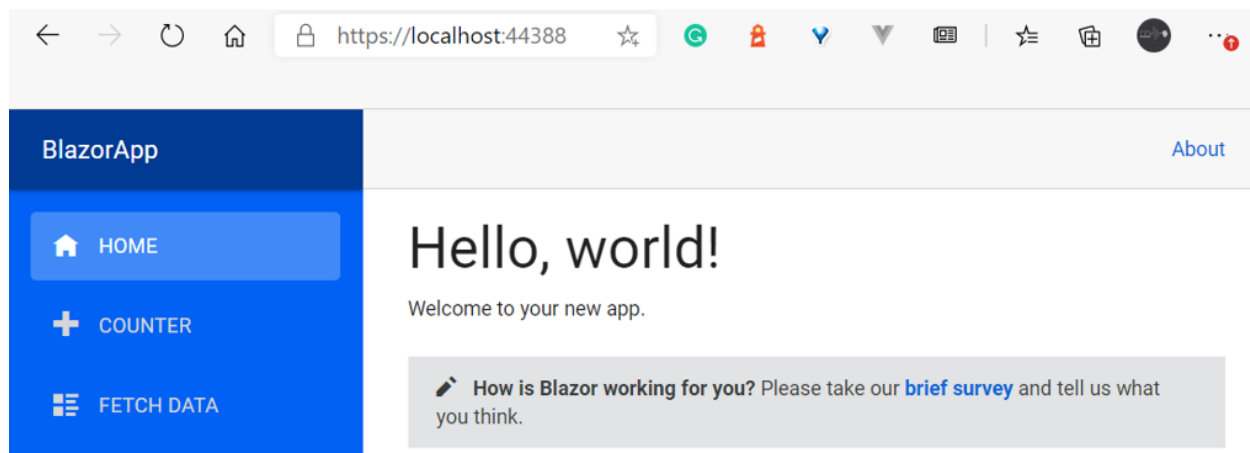


Figure 1-i: The App Running (with Hot Reload, Before Any UI Changes)

Look at that! With the changes we've made so far, we can see that the application has a [Material Design](#) look and feel.

Let's make a change to the UI to see the hot reload functionality in action. Open **Solution Explorer**, navigate to the **Pages** folder, and then open the **Index.razor** file.

Within the **h1** section, change the text from "**Hello, world!**" to "**Hi, world!**". Then, in Visual Studio, press **Ctrl+S** to save the change.

If you look at the browser window, the change should be visible. In my case, it appears as follows.

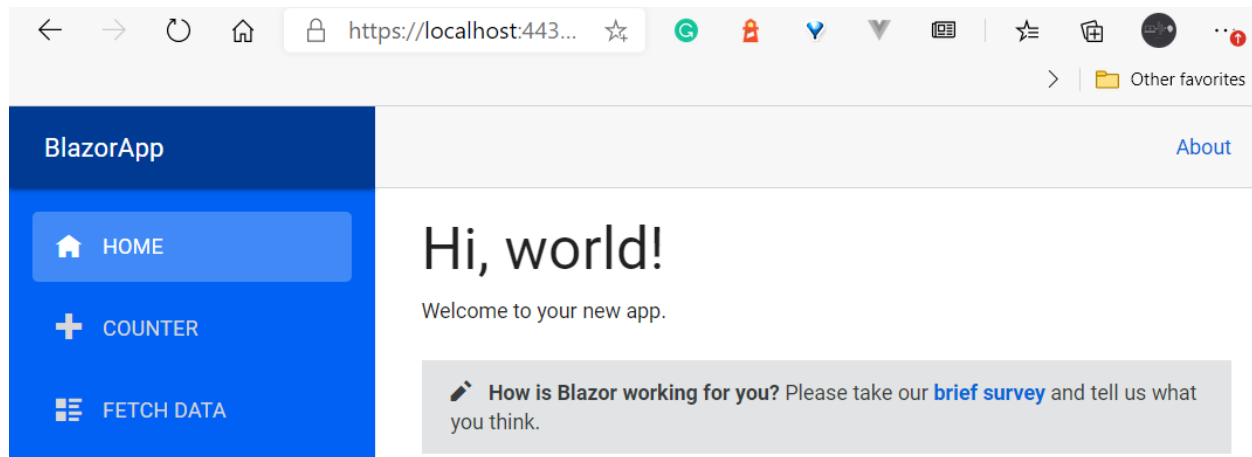


Figure 1-j: The App Running (with Hot Reload, After the Change)

Awesome! Not only do we have a nice-looking Blazor Server app, but we also have hot reloading working in Visual Studio 2019.

Summary

Alright—we've reached the end of this chapter and now have a basic Blazor Server application we can work with.

In the next chapter, we will start by writing a basic component and then build up our knowledge by looking at how components are rendered and how to organize components into class libraries.

After that, we will look at how code is structured within components, and how components handle events, child content, and one-way data binding.

Chapter 2 Component Fundamentals

Having bootstrapped and made some changes to our Blazor Server application, we are now ready to move our attention to Razor components and how to work with them.

Throughout this chapter, we will cover the fundamental aspects of components, how they work, and how they are rendered, so that we can start to add them to our bootstrapped application.

Component basics

Looking at our bootstrapped application, note that the left sidebar is a component called **NavMenu**. It is highlighted in the following figure.

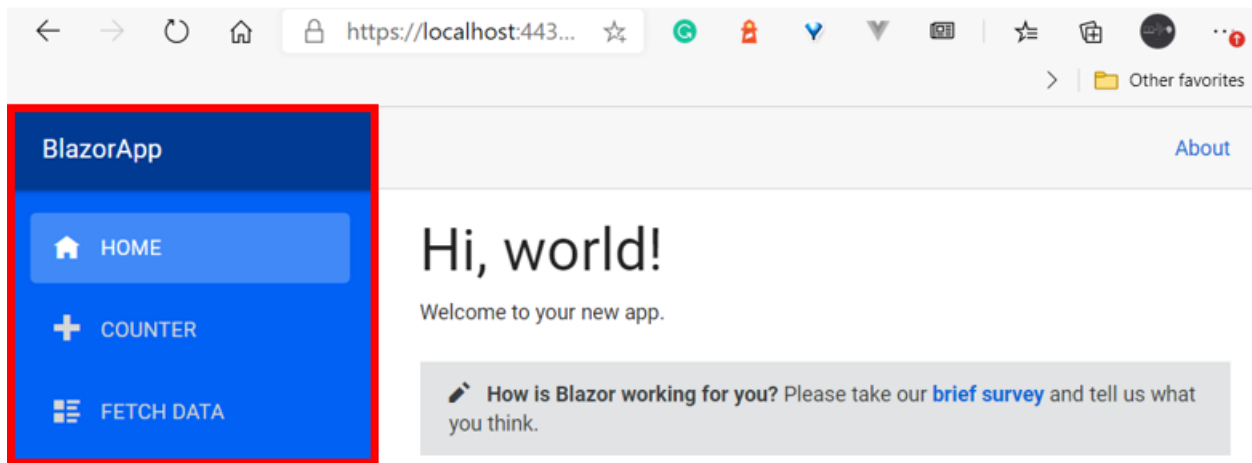


Figure 2-a: The NavMenu Component (Rendered in the App's UI)

This component is defined within the project structure and can be easily found using **Solution Explorer**. It resides in the **Shared** folder of the project.

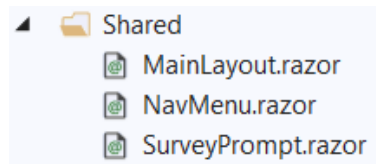


Figure 2-b: The NavMenu Component (Project Structure)

By exploring the project structure, we can see that our application has components residing in two folders: **Pages** and **Shared**.

The **Pages** folder contains components that are rendered as webpages within our application. The **Shared** folder contains components that can be used by multiple pages.

For example, the **Index.razor** page component renders the application's main page (also known as the index page) as seen in Figure 2-a.

If we open the **Index.razor** file, we'll notice two things. The first thing is that it starts with a **@page** directive—this tells Blazor that this is a page component.

The second thing to notice is that there's a **SurveyPrompt** tag being referenced in the HTML markup—this tag refers to a separate component with the same name.

Let's have a look at the source code of the **Index.razor** file.

Code Listing 2-a: Index.razor

```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

So, what is happening here? The **@page** directive is followed by a relative web address, which is the route of the webpage (/).

Next, we find the **h1** tag, followed by some text, and after, the reference to the **SurveyPrompt** component.

By referencing the **SurveyPrompt** component within the index page, we are extending the HTML markup within the **Index.razor** file, given that **SurveyPrompt** is now an extended HTML tag.

So, one way to think of components within Blazor is that they are a way to extend HTML and reuse code within different parts of the application.

Let's quickly have a look at the source code of the **SurveyPrompt** component. To do that, go to **Solution Explorer** and open the **SurveyPrompt.razor** file, which resides in the **Shared** folder.

Code Listing 2-b: SurveyPrompt.razor

```
<div class="alert alert-secondary mt-4" role="alert">
  <span class="oi oi-pencil mr-2" aria-hidden="true"></span>
  <strong>@Title</strong>

  <span class="text-nowrap">
    Please take our
    <a target="_blank" class="font-weight-bold"
      href="https://go.microsoft.com/fwlink/?linkid=2112271">
      brief survey</a>
```

```

    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters.
    [Parameter]
    public string Title { get; set; }
}

```

As we can see, the source code for the **SurveyPrompt** component contains the HTML markup part and the C# code part, which is distinguishable by the **@code** directive.

The HTML markup contains a **div** with **span** tags, text, and a link to an external survey page. Notice that the **Title** property is referenced within the HTML markup.

The **Title** property within the markup binds to a C# property with the same name, which is passed as a parameter to the **SurveyPrompt** component when it is referenced from the **Index.razor** file as follows:

```
<SurveyPrompt Title="How is Blazor working for you?" />
```

Within Razor files, it is possible to combine HTML markup, CSS styles, and also C# code. Other web application frameworks such as Vue also allow developers to combine into a single file HTML markup, styles, and code.

We'll explore this topic further a bit later, as well as how to best organize our component's code.

Component reusability

A component is just a reusable piece of code or UI. Reusability is not only something that helps applications scale, but it also helps the long-term maintainability of an application.

So, the simplest way to demonstrate code reusability in our application is to reference the **SurveyPrompt** component within another page.

Let's reference the **SurveyPrompt** component within the **Counter.razor** file. To do that, go to **Solution Explorer** and open the **Counter.razor** file, which resides in the **Pages** folder.

Once you open the file, you should see the following code.

Code Listing 2-c: Counter.razor

```

@page "/counter"

<h1>Counter</h1>

```

```

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}

```

Now, let's copy the following line and paste it after the `</button>` tag:

```
<SurveyPrompt Title="How is Blazor working for you?" />
```

The code should now look as follows. The added line is highlighted in bold.

Code Listing 2-d: Counter.razor (with the SurveyPrompt Component Added)

```

@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

<SurveyPrompt Title="How is Blazor working for you?" />

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}

```

If we now run the application by pressing **Ctrl+F5**, we should see the following result after clicking the **COUNTER** option in the app's navigation menu.

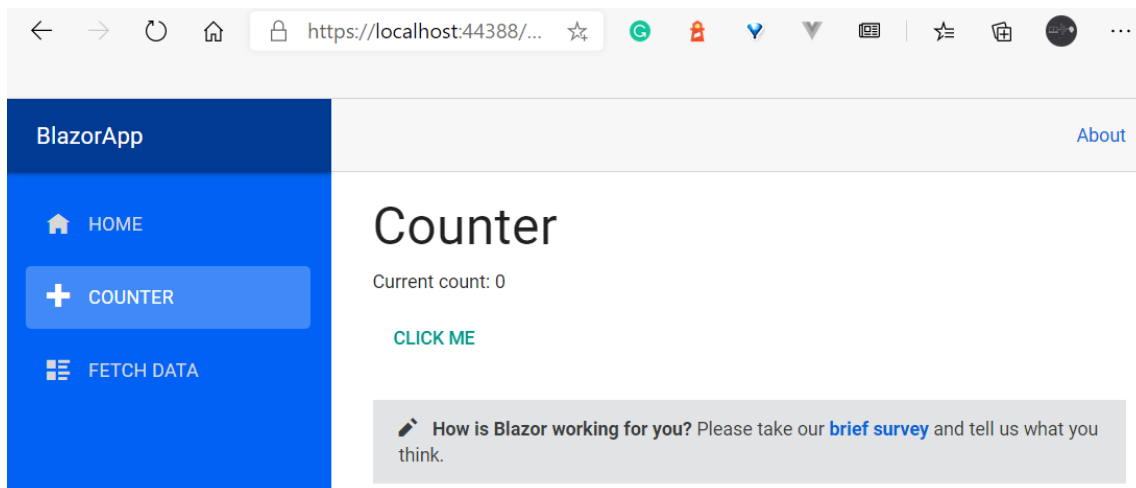


Figure 2-c: The Updated Counter Page (Using the SurveyPrompt Component)

By doing this, we have reused the **SurveyPrompt** component, which we have embedded within the **Index.razor** and **Counter.razor** files.

Creating a new component

Now that we've seen how we can reuse an existing component, let's create a new one that we can embed in the **Counter.razor** file.

So, let's go to **Solution Explorer**, right-click the **BlazorApp** project name, click **Add**, and then select **New Folder**. Once done, name the folder **Components**.

Right-click on the **Components** folder, click **Add**, and then click **Razor Component**. Give the component file a name—for example, **TestButton.razor**.

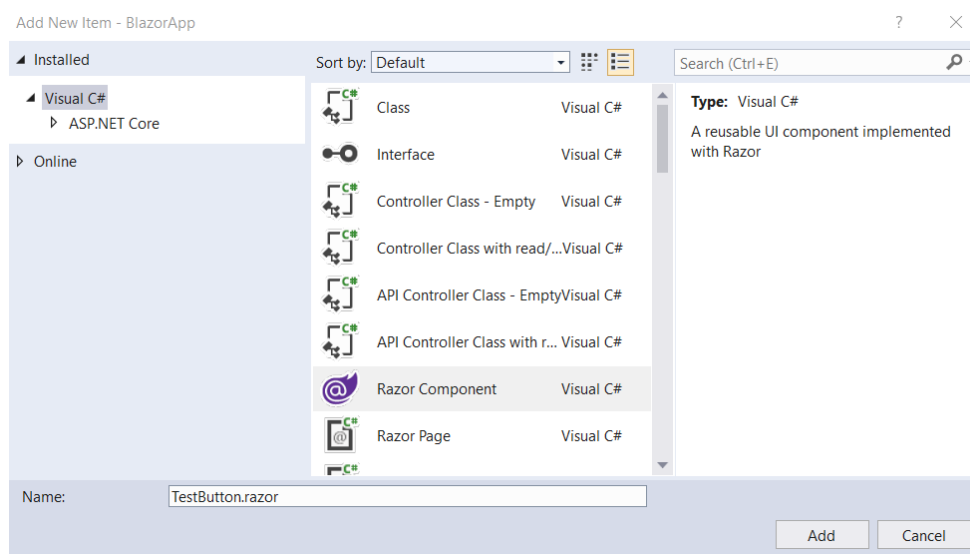


Figure 2-d: Adding a New Razor Component

To finish creating the component, click **Add**. Once the **TestButton.razor** file has been created, open the file and replace the existing code with the following.

Code Listing 2-e: TestButton.razor (With New Code)

```
<Button Class="custom-css"
        Style="margin: 10px;"
        Variant="@ButtonVariant.Contained"
        Color="@Color.Primary">
    This is a Test button :)
</Button>

@code {
}
```

Congrats! We've just created our first component. However, we cannot visualize it yet, as this component is not part of any of our app's pages.

To visualize it, we need to embed it in one of the existing pages, so let's add it to the **Counter.razor** file.

But before do that, we need to reference the **BlazorApp.Components** namespace in the **_Imports.razor** file, as follows. The change is highlighted in bold.

Code Listing 2-f: Adding BlazorApp.Components to _Imports.razor

```
// Previous references...

@using BlazorApp.Shared
@using BlazorApp.Components

// Further references...
```

By doing this, we can embed and use the **TestButton** component in any part of our Blazor application.

So now, let's go to **Solution Explorer**, open the **Counter.razor** file that resides within the **Pages** folder, and add the following line of code, just before the **@code** directive.

```
<TestButton />
```

The updated source code for the **Counter.razor** file should look like the following code listing. The change is highlighted in bold.

Code Listing 2-g: Updated Counter.razor

```
@page "/counter"
```

```

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

<SurveyPrompt Title="How is Blazor working for you?" />
<TestButton />

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}

```

If you previously executed the application by pressing **Ctrl+F5** and the app is still visible in the browser, then hot reloading should be working. If you click on the **COUNTER** option on the app's navigation menu, you should see some changes.

If not, run the application again by pressing **Ctrl+F5**. Once the app is running, you should see the new button below the **SurveyPrompt** component.

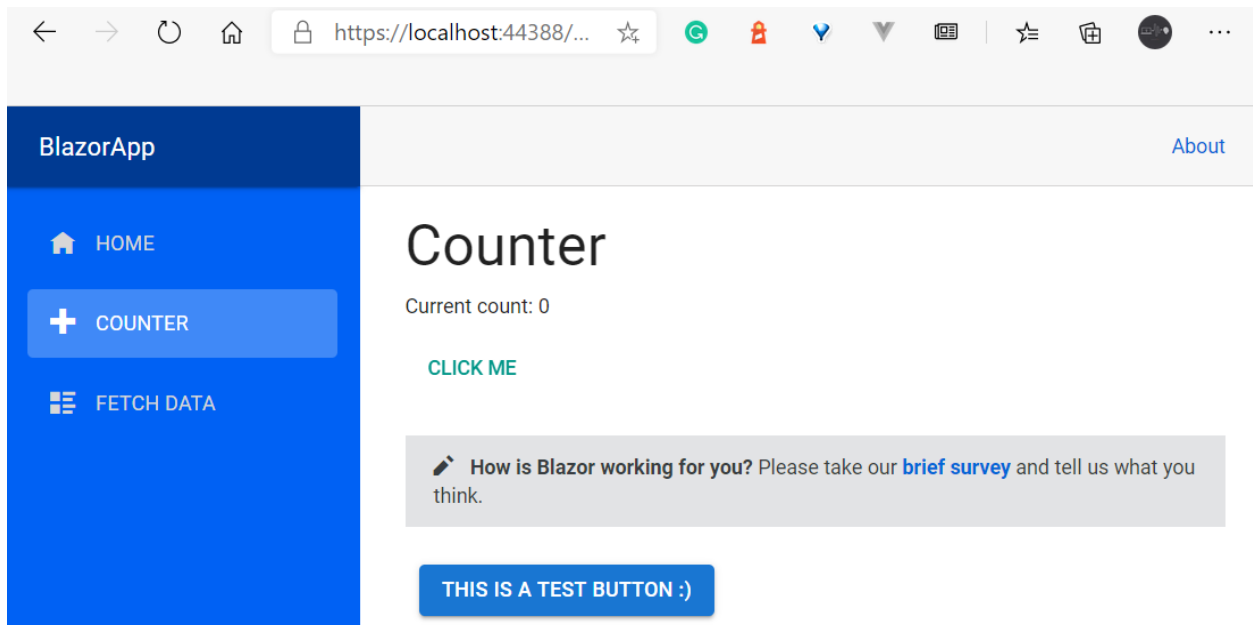


Figure 2-e: The New Button Component (Counter.razor)

Awesome—we've now seen how to add a new component to our Blazor Server application.

How components are rendered

The Blazor framework comes in two flavors. One is called Blazor Server, where the components are rendered on the server and a SignalR connection is used to communicate between the server and client. The following figure illustrates how the Blazor Server model works.

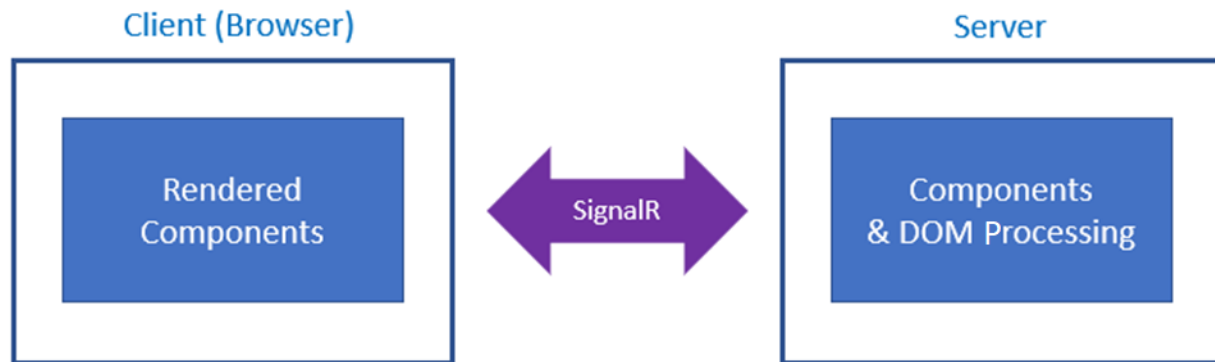


Figure 2-f: The Blazor Server Model

Blazor Server apps are hosted on an [ASP.NET Core](#) server using the [ASP.NET Razor syntax](#). Using this model, web browsers act as thin clients, meaning that the bulk of the processing load is done on the server.

The client's web browser downloads a small page and updates its UI over a SignalR connection.

The second flavor of the framework is Blazor WebAssembly, where C# code is sent to the client (browser) to be interpreted.

To do that, Blazor ensures the browser has a runtime that is created using WebAssembly, which is a binary format designed to run very fast in browsers. The following figure illustrates how the Blazor WebAssembly model works.

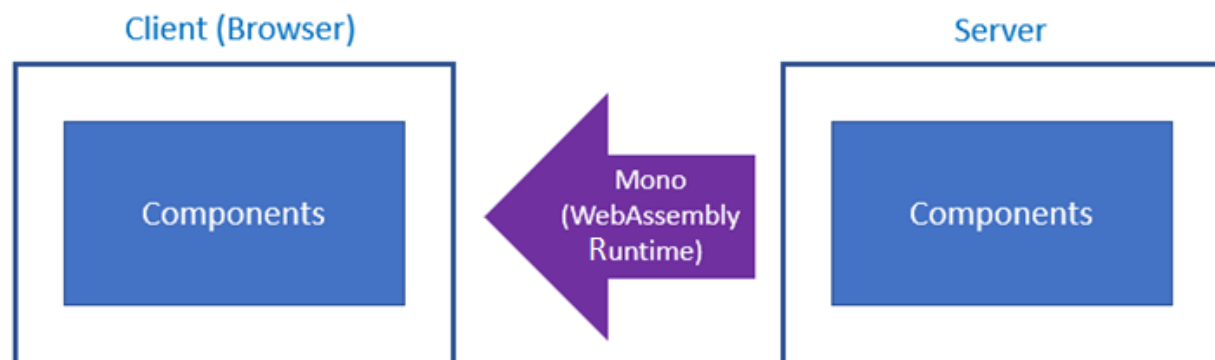


Figure 2-g: The Blazor WebAssembly Model

Using the Blazor WebAssembly model, single-page apps are downloaded to the client's web browser before running.

The size of the download is larger than that of Blazor Server apps, and the processing is entirely done on the client hardware.

Using this model, the application type typically enjoys faster response times than using the Blazor Server model. This client-side framework is written in WebAssembly, as opposed to JavaScript.

A key element to understand is that although these models of Blazor are different from one another, they both execute an application by rendering a root component called **app**.

The **app** component is referenced within the **_Host.cshtml** file, which resides in the **Pages** folder. For our Blazor Server application, the **app** component is described as follows.

```
<app>  
  <component type="typeof(App)" render-mode="ServerPrerendered" />  
</app>
```

Notice that **render-mode** is set to **ServerPrerendered**, which means that the components are prerendered on the server and sent to the client through a SignalR connection.

Therefore, the **app** is always the root component of your application, regardless of the Blazor model you use.

The **app** component is identical for both the Server and WebAssembly versions of Blazor, and all the components that are part of the application will be rendered through this root component.

Therefore, the **app** component can render child components, and these child components can render other child components.

This means that a Blazor application's structure is made up of pages, which are made up of one or more components, and these components are made up of other child components.

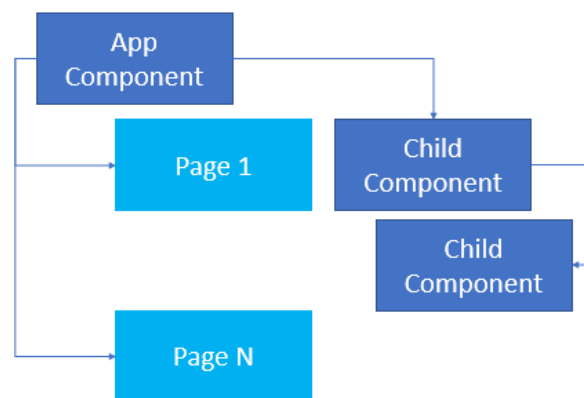


Figure 2-h: Blazor App Structure

Now that we have explored how components are rendered in Blazor, it's time to look at some of the differences between Blazor Server components and Blazor WebAssembly components.

Component differences between models

What components can or cannot do differs between Blazor Server apps and Blazor WebAssembly apps.

For example, say that in our Blazor Server app we create a component that is responsible for accessing the file system.

On the server, that component will have access to the file system of the server. Later, we want to use that same component in a Blazor WebAssembly application.

At first, you might naturally think that because this component has access to the local file system of the server, this component will have access to the local file system of the client when it is integrated into a Blazor WebAssembly application.

However, that's not the case. Giving the browser access to the file system would be a major security risk, so that won't work.

So, from a functional perspective, the component responsible for accessing the file system, which worked great in the server application, won't work in the client app.

Nevertheless, the way the component is structured internally, and the component's features, are identical for both Blazor Server and WebAssembly applications.

All the topics covered in this book will be for the Blazor Server application we previously bootstrapped.

However, because the book is about component features and not details that are specific to any of the Blazor models, almost everything will apply to both Blazor Server and WebAssembly apps, one-to-one.

Page routing and rendering

The **App.razor** file that resides within the project's root folder is referred to as the app Router component. Let's have a look at its code.

Code Listing 2-h: App.razor

```
<!--New-->
<MaterialScripts />
<MaterialStyles />

<!--New-->
```

```
<ThemeProvider Theme="@Theme.Light" />

<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData"
      DefaultLayout="@typeof(MainLayout)" />
  </Found>
  <NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
      <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
  </NotFound>
</Router>
```

We can see that within the **App.razor** file, there is a reference to the **Router** component and some other components that are part of the Blazor framework, such as **Found**, **NotFound**, and **RouteView**.

The **Router** component doesn't render any UI element by itself, so its purpose is to look within the given assembly for pages—which, as you know, are Blazor components with the **@page** directive, which contain routing information.

If a match between the URL and the routing configuration of a page is found, the **RouteView** component is used to render the page, passing along the **routeData** and layout page (**DefaultLayout**) to be used. This can be better understood by looking at the following diagram.

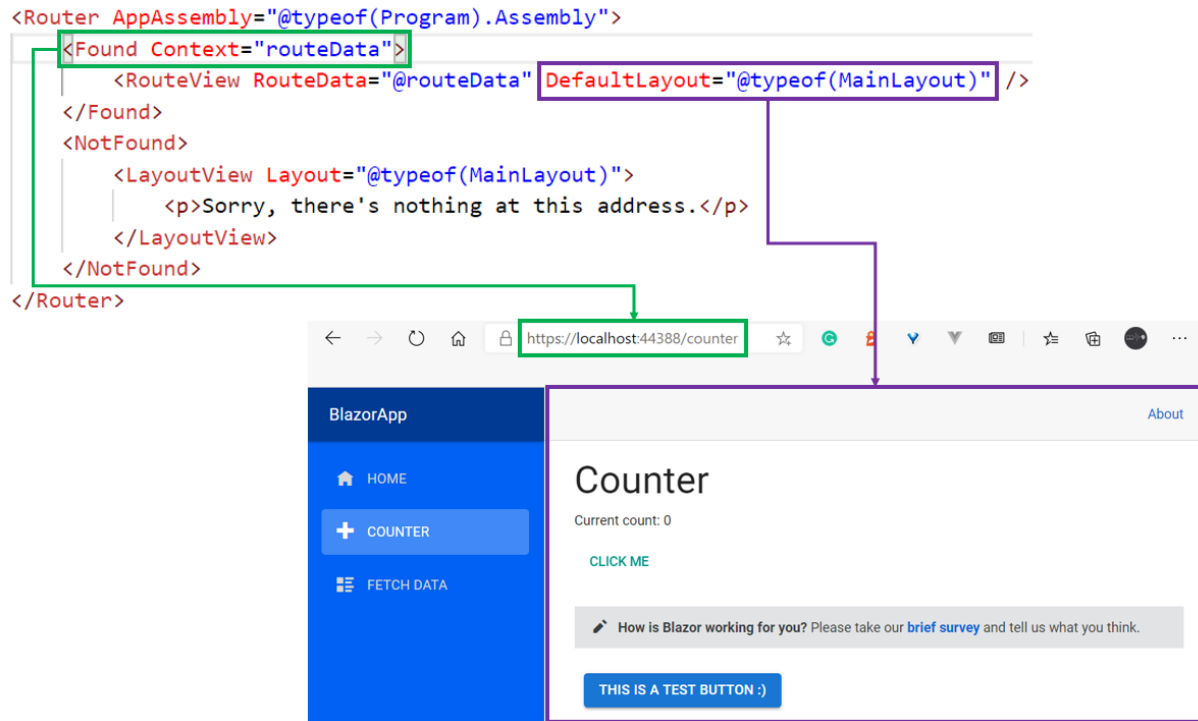


Figure 2-i: How the Router Component Renders a Page (If a Match Is Found)

On the contrary, if a match is not found, then a **LayoutView** component is used, which takes a page and renders the HTML markup provided as the body for it.

In the following diagram, we can see that the URL points to a page called **bla**, which doesn't exist. Therefore the request gets routed to the **NotFound** component.

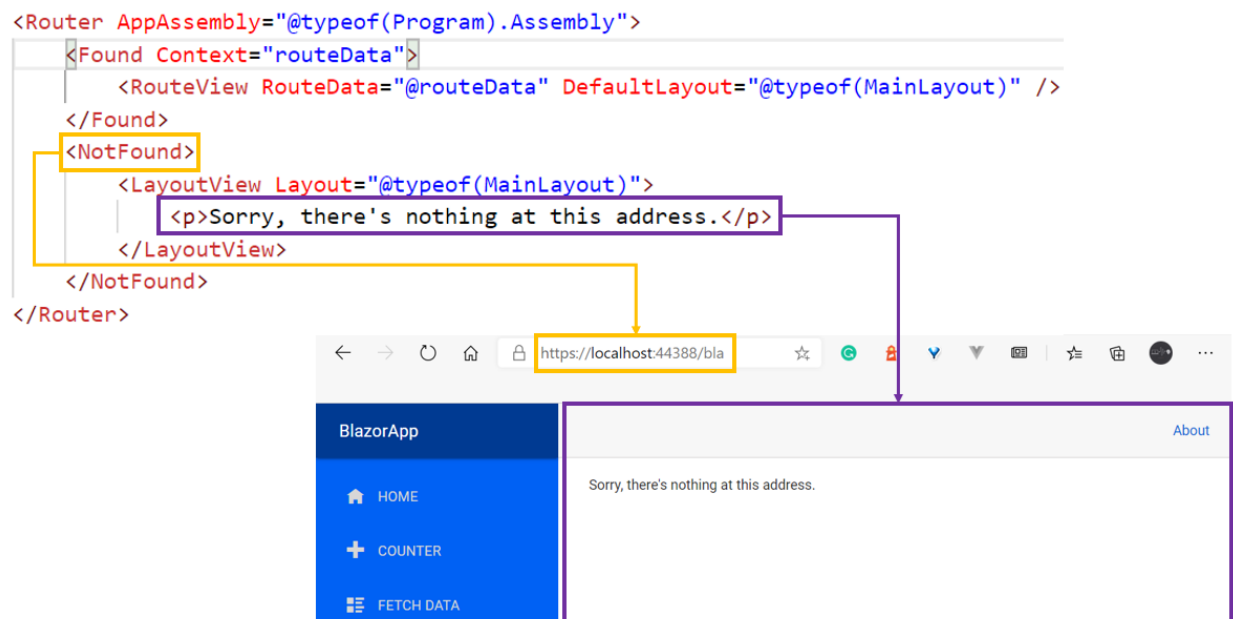


Figure 2-j: How the Router Component Renders a Page (If a Match Is Not Found)

Now, let's say that a new request comes in at the root. The **Router** component will search for a page component that has a **@page** directive specifying root, which in this case is the page component contained within the **Index.razor** file. The following diagram illustrates this.

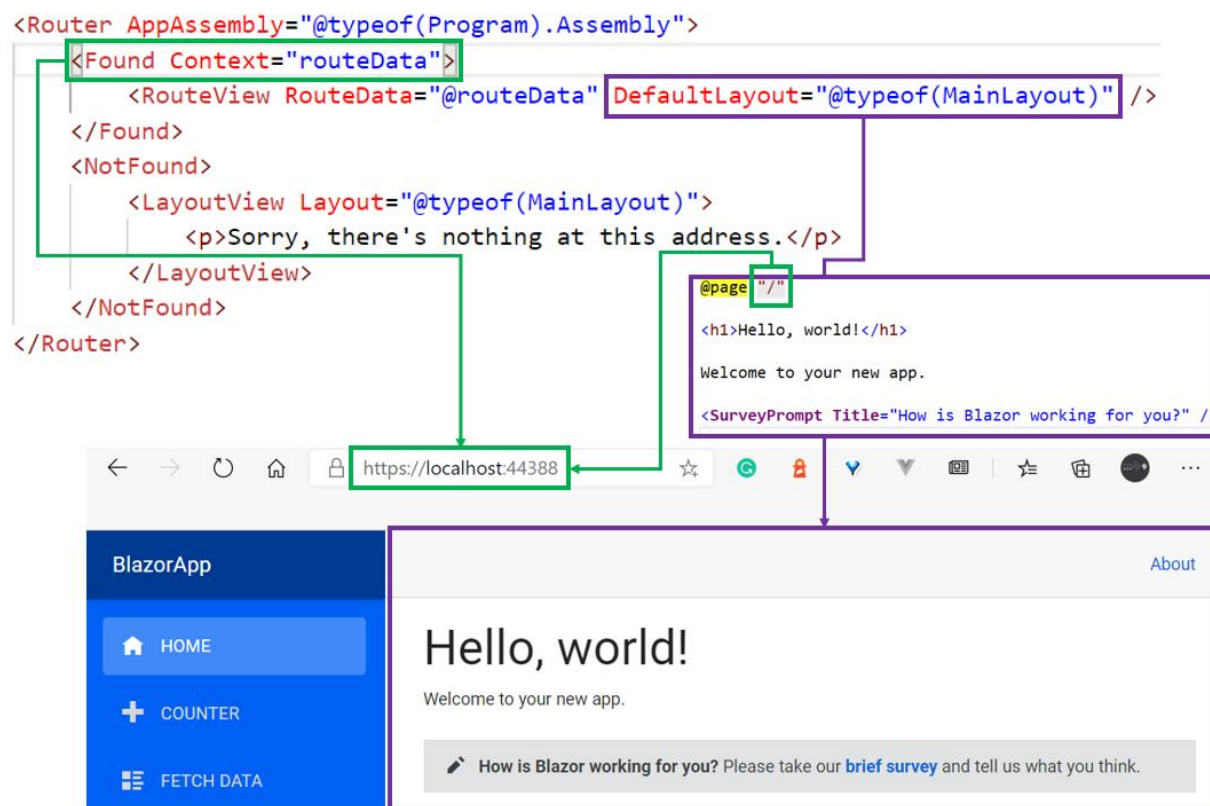


Figure 2-k: How the Router Component Renders a Page (Accessing the Site's Root Page, *Index.razor*)

As we can see in the **Router** component, when a route is found, every page rendered will use **MainLayout.razor** to render the page's child content (the content of the page to be rendered) at the location marked with the **Body** property.

What this means in practice is that if **Index.razor** is the page component that is displayed, its markup content is injected into the section of **MainLayout.razor** that includes the **Body** property. The following diagram illustrates this process.

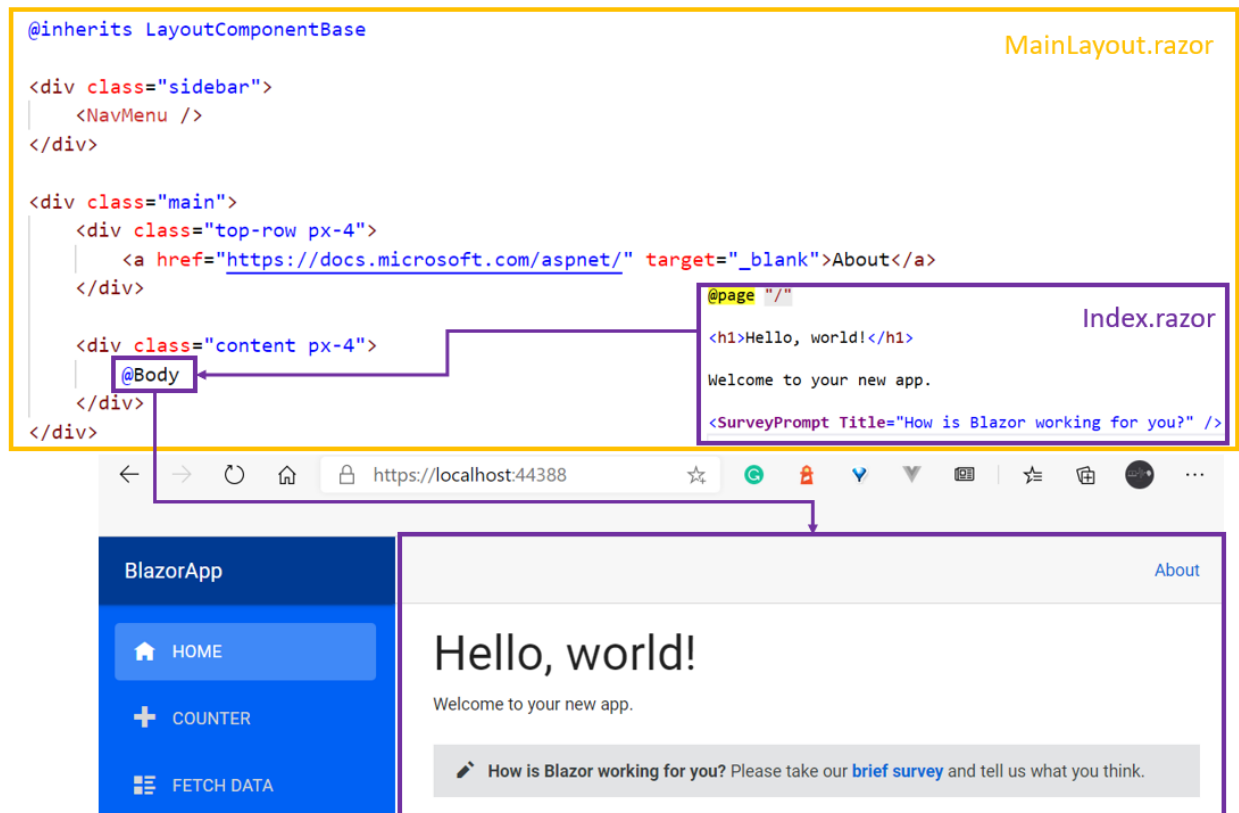


Figure 2-1: How a Page Is Injected into MainLayout.razor

We can see how the **MainLayout.razor** file also includes a reference to the **NavMenu** component, which corresponds to the blue menu found on the left-hand side of the application's UI.

Now that we know how the app's components are rendered, let's move our attention next to how we can organize our component's code better.

Component code organization

One of the great things about files using the **.razor** extension is that HTML markup and C# code can be combined into a single file. For someone who's starting fresh in Blazor or web development, this is quite appealing, as it means there are fewer files to maintain within a project.

Not only can you combine HTML markup and C# code into a single file, but you can have multiple code segments in your component, which is very powerful.

Despite the power and apparent simplicity of these capabilities, when components become more complex, it might not be such a good idea to have the HTML markup and C# code combined into a single file, which is an opinion that some people might share and some might not.

My approach to code organization when working and writing components is that the code should not be mixed with the HTML markup. As a component grows in complexity, having a separation of concerns between the HTML and C# code is something that you might find useful and come to appreciate as your codebase expands.

A way to achieve this is by giving the component file a companion, known as a code-behind file. This can be done by going to **Solution Explorer** and creating a new C# class file next to the component file.

Let's do that for the **TestButton.razor** file that resides in the **Components** folder. So, right-click on the **Components** folder name within **Solution Explorer**, and then click **Add**, followed by **New Item**. From the list, choose **Class**.

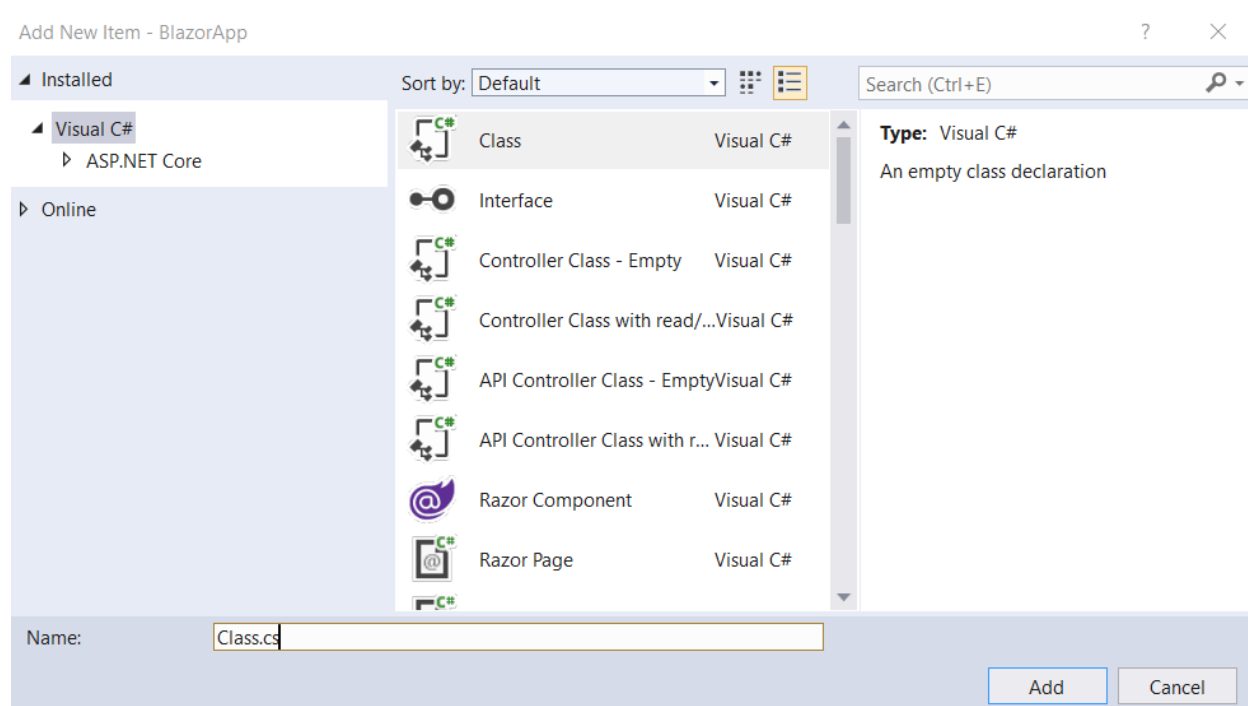


Figure 2-m: Creating a Code-Behind File (Visual Studio)

Let's give the class a name, which in theory can be any name. However, it is good to follow a certain naming convention, which will help make things easier as we go. A particularly good convention to follow is to give the class the name of the component, followed by the **Base** suffix (or prefix if you prefer; whichever you choose, be consistent with the naming convention when trying this out).

So, in this case, we would name this class file **TestButtonBase.cs**, as can be seen in the figure that follows.



Figure 2-n: Code-Behind File Name (Visual Studio)

Once the file has been created, we can see it in **Solution Explorer** as follows.

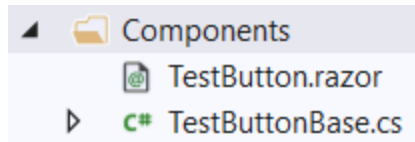


Figure 2-o: The TestButtonBase.cs File (Components Folder)

Now, to understand why we are doing this, there's one thing to be aware of that is not particularly obvious to someone who's coming newly to Blazor from another framework.

Hidden code-behind

Every component file within the project has a C# code-behind file that is not visible (i.e. not shown within the Visual Studio solution).

We can see this by going into **Solution Explorer**, and at the project level (which has the **BlazorApp** name on it), right-click and then click on the **Open Folder in File Explorer** option. This will open the folder where the project resides on disk.

Once you do that, in the folder search box, type ***g.cs**, and shortly after you should get some results. Scroll down and locate the **TestButton.razor.g.cs** file shown in the following figure.

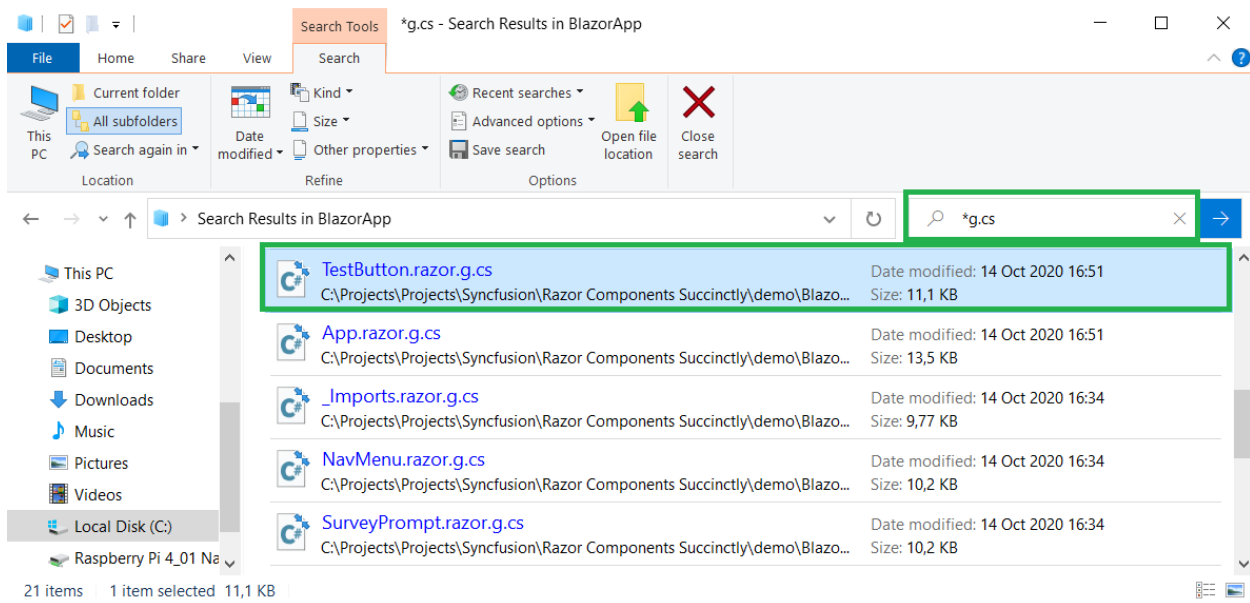
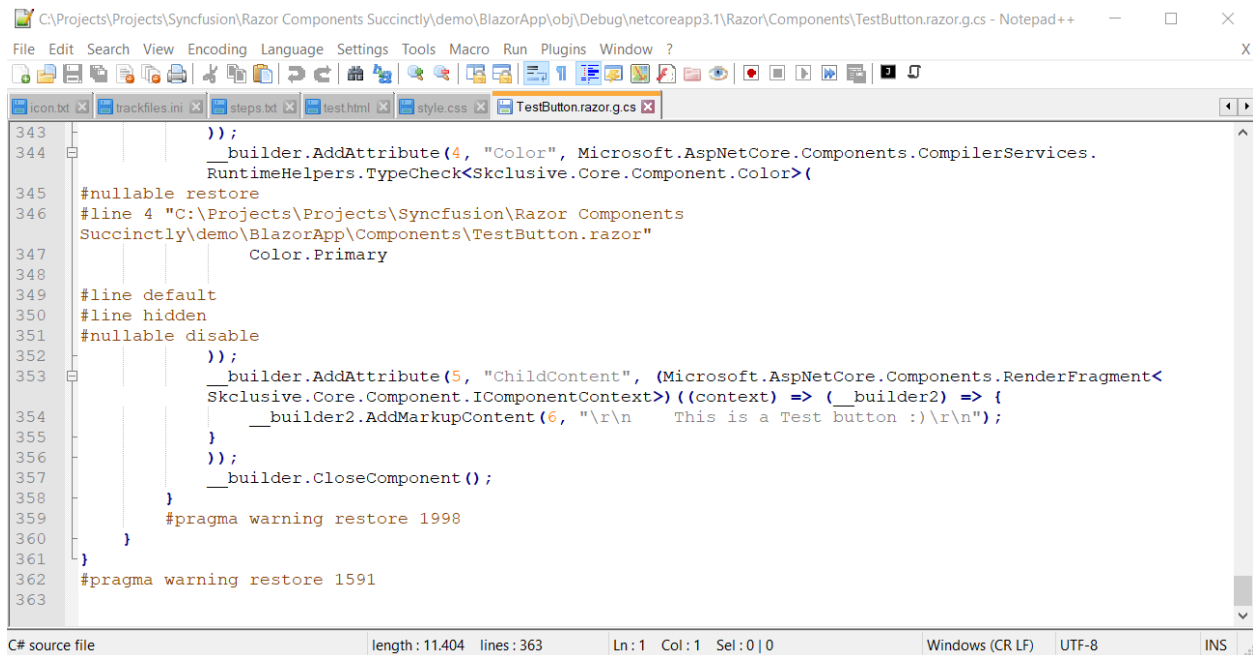


Figure 2-p: Hidden Code-Behind Files (Project Folder, Windows Explorer)

Just out of curiosity, let's open this file with Notepad, [Notepad++](#), or any other text editor of your preference to see what the code looks like.

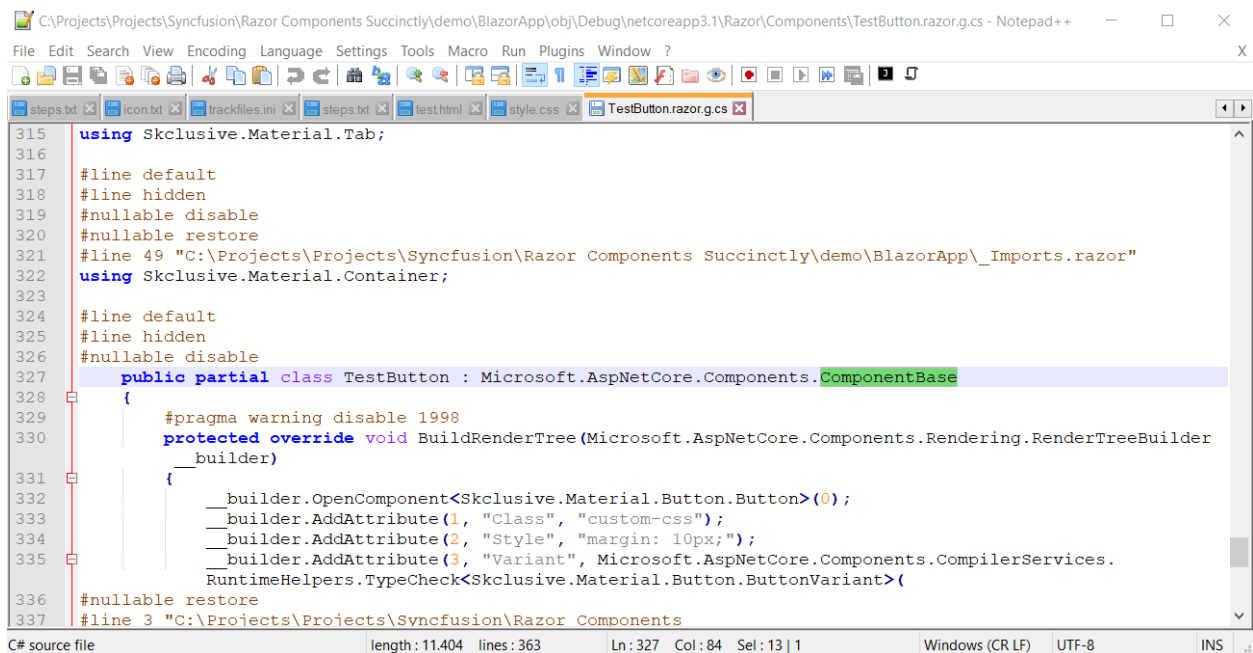


```
343
344     builder.AddAttribute(4, "Color", Microsoft.AspNetCore.Components.CompilerServices.
        RuntimeHelpers.TypeCheck<Skclusive.Core.Component.Color>()
345 #nullable restore
346 #line 4 "C:\Projects\Projects\Syncfusion\Razor Components
        Succinctly\demo\BlazorApp\Components\TestButton.razor"
347         Color.Primary
348
349 #line default
350 #line hidden
351 #nullable disable
352
353     builder.AddAttribute(5, "ChildContent", (Microsoft.AspNetCore.Components.RenderFragment<
        Skclusive.Core.Component.IComponentContext>)((context) => {
354         __builder2.AddMarkupContent(6, "\r\n    This is a Test button :)\r\n");
355     });
356
357     __builder.CloseComponent();
358 }
359 #pragma warning restore 1998
360 }
361 }
362 #pragma warning restore 1591
363
```

Figure 2-q: The TestButton.razor.g.cs Hidden Code-Behind File in Notepad++

If you scroll down, you'll notice that this file was machine-generated and it is not very easy to read or follow, even though it is C# code.

There's one very important thing to note: the generated **TestButton** class inherits from the **ComponentBase** class, which in my case can be seen in the following figure, on line 327.



```
315 using Skclusive.Material.Tab;
316
317 #line default
318 #line hidden
319 #nullable disable
320 #nullable restore
321 #line 49 "C:\Projects\Projects\Syncfusion\Razor Components Succinctly\demo\BlazorApp\_Imports.razor"
322 using Skclusive.Material.Container;
323
324 #line default
325 #line hidden
326 #nullable disable
327 public partial class TestButton : Microsoft.AspNetCore.Components.ComponentBase
328 {
329     #pragma warning disable 1998
330     protected override void BuildRenderTree(Microsoft.AspNetCore.Components.Rendering.RenderTreeBuilder
        __builder)
331     {
332         __builder.OpenComponent<Skclusive.Material.Button.Button>(0);
333         __builder.AddAttribute(1, "Class", "custom-css");
334         __builder.AddAttribute(2, "Style", "margin: 10px;");
335         __builder.AddAttribute(3, "Variant", Microsoft.AspNetCore.Components.CompilerServices.
        RuntimeHelpers.TypeCheck<Skclusive.Material.Button.ButtonVariant>()
336 #nullable restore
337 #line 3 "C:\Projects\Projects\Syncfusion\Razor Components
```

Figure 2-r: The Generated Class (Inheriting from ComponentBase)

So, you might be asking yourself—what does all this have to do with the fact that we have created a new file called **TestButtonBase.cs**?

Component inheritance

To get a hint of why we are doing this, let's open the **TestButtonBase.cs** file with **Solution Explorer** by double-clicking on it and viewing its content.

Code Listing 2-i: The New TestButtonBase.cs File (Source Code)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace BlazorApp.Components
{
    public class TestButtonBase
    {
    }
}
```

There's nothing particularly interesting about the previous code. Have you guessed where we are going with this? If so, great. If not, no worries. Let's take a step back and analyze this from a different angle.

The reason for creating a **TestButtonBase.cs** file (adding the **Base** suffix to the name **TestButton**), is that we are going to organize the C# code for the **TestButton** component in a way that this component inherits from the **TestButtonBase** class, rather than **ComponentBase**, as it is now. Let's look at the following diagram to understand how the class hierarchy is now.

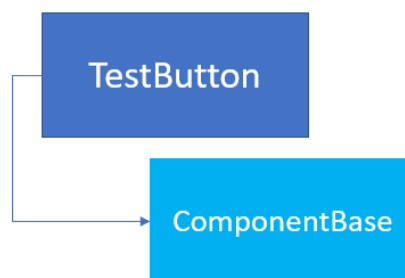


Figure 2-s: Out-of-the-Box TestButton Component Inheritance

So, what do we do with the **ComponentBase** class inheritance? Well, you might have guessed by now, but let's look at the following diagram first.

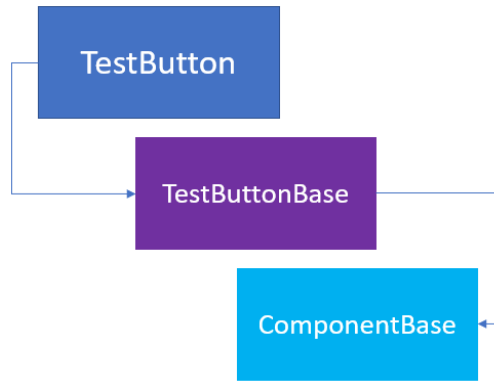


Figure 2-t: The New TestButton Component Inheritance

In essence, the **TestButtonBase** class we created with the **TestButtonBase.cs** code-behind file must inherit from the **ComponentBase** class for everything to work—so, let's add that to our code. The changes are highlighted in bold.

Code Listing 2-j: The New TestButtonBase.cs File (Updated Source Code)

```
using Microsoft.AspNetCore.Components;

namespace BlazorApp.Components
{
    public class TestButtonBase : ComponentBase
    {
    }
}
```

Notice that the **ComponentBase** class resides within a specific namespace, so that's why I have added it with the following statement.

```
using Microsoft.AspNetCore.Components;
```

Another thing we can do is rename the file from **TestButtonBase.cs** to **TestButton.razor.cs**.

If the [file nesting](#) option is enabled within **Solution Explorer** in Visual Studio, the **TestButton.razor.cs** file will appear under the **TestButton.razor** file, as shown in the figure that follows.

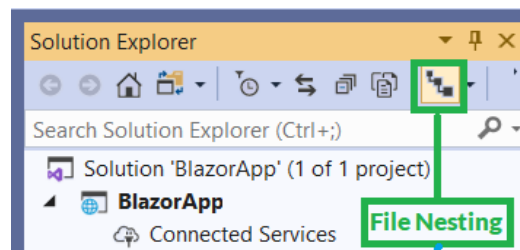


Figure 2-u: The File Nesting Option Enabled in Visual Studio (Solution Explorer)

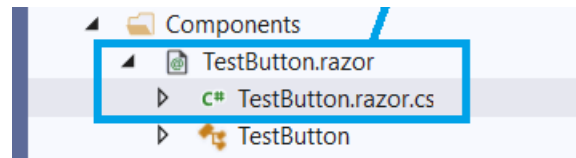


Figure 2-v: *TestButton.razor.cs Nested Under TestButton.razor*

So, the purpose of this exercise and using either approach, **TestButtonBase.cs** or **TestButton.razor.cs**, is that we want to keep the HTML markup free from C# code, so if our component becomes too complex in the future, all the code is easier to read and maintain.

Razor file nesting

Because I like the file nesting feature that **Solution Explorer** provides, I prefer to use and keep the **TestButton.razor.cs** file rather than **TestButtonBase.cs**, but that's just my personal preference. From my point of view, it keeps the C# code-behind nicely organized under the HTML markup code file.

You are free to use **TestButtonBase.cs** if you prefer that convention going forward, and add the **Base** suffix to other components. I'll use the nested option instead.

I went through this exercise and created the **TestButtonBase.cs** file to show you how to get to the **TestButton.razor.cs** file and add that extra layer of inheritance needed for C# code-behind to work.

So, to wrap this up for the **TestButton** component, there are two things we still need to do. First, we need to remove the **@code** directive and code block from the **TestButton.razor** file.

Second, we need to make sure that the **TestButton** component within the **TestButton.razor** file inherits from the **TestButtonBase** class. To do that, we need to add the **@inherits** directive to the code of the **TestButton.razor** file.

We can see the updated code of the **TestButton.razor** file in the following code listing.

Code Listing 2-k: *The Updated TestButton.razor Code*

```
@inherits TestButtonBase

<Button Class="custom-css"
        Style="margin: 10px;"
        Variant="@ButtonVariant.Contained"
        Color="@Color.Primary">
    This is a Test button :)
</Button>
```

If we now run the application in Visual Studio by pressing **Ctrl+F5**, and then navigate to the **Counter** page, we should see the following.

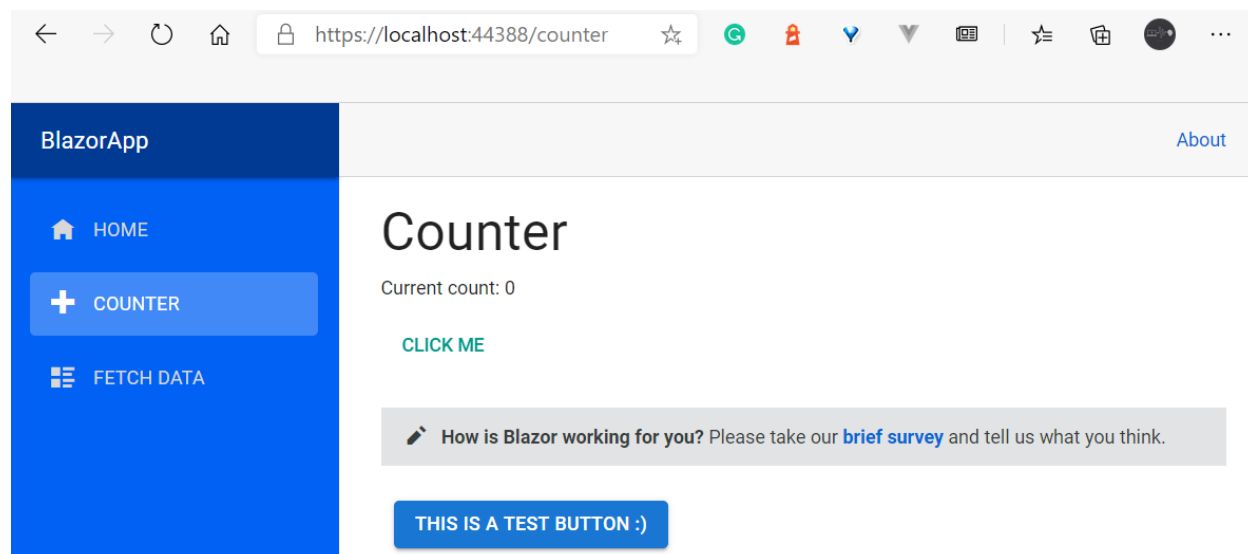


Figure 2-w: The Counter Page

So, from a UI perspective, nothing has changed. All we have done is organize our component's code differently using a code-behind approach with additional class inheritance.

Perhaps the greatest advantage of this approach—using inherited classes as the code-behind for components—is that you can add as many class layers as needed, especially when there's common functionality for various components.

Partial classes

There's also another way to create code-behind component classes, which is to use partial classes. By using partial classes, we can create part of the class in another file. During compilation, partial classes in different files will be merged into a single class. Placing a class definition in two files is useful when you intend for one part of the definition to never be edited, and the other part to be edited if needed.

Let's create a partial class for our **TestButton** component and try that instead of using the previous approach.

Within **Solution Explorer**, click on the **Components** folder, right-click, then click **Add**, select **New Item**, and choose **Class** from the list of items. Let's call it **TestButtonPartial.cs** and then click the **Add** button.

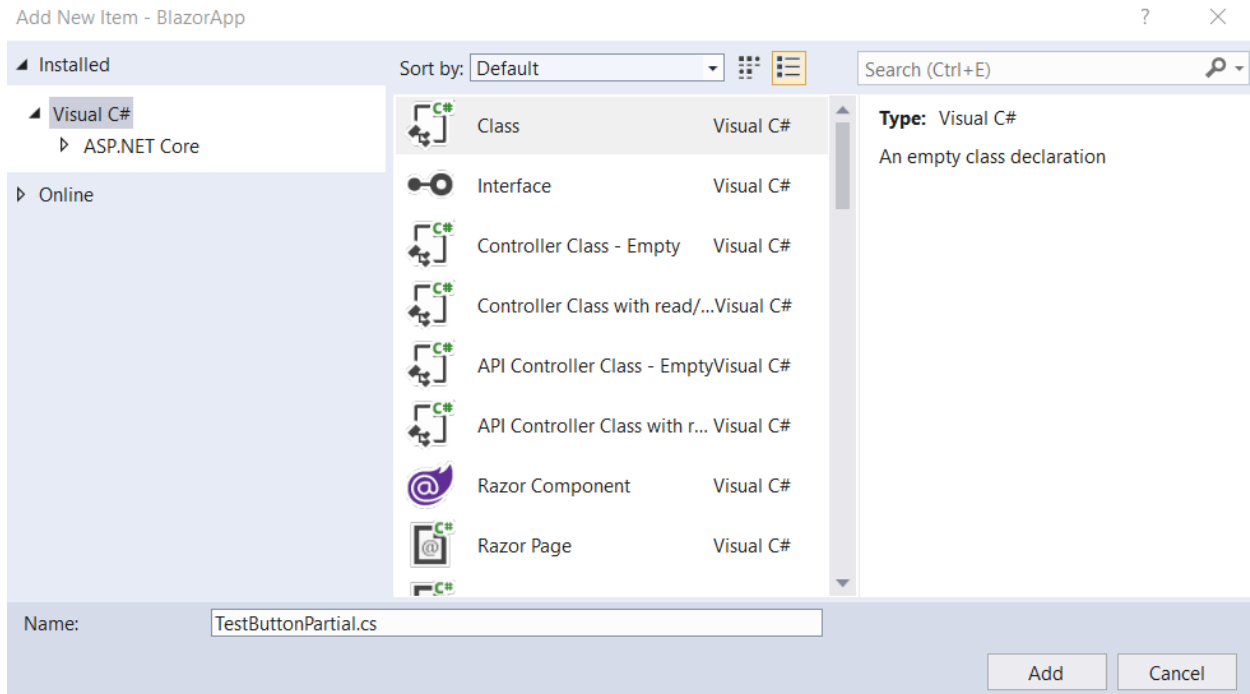


Figure 2-x: Adding *TestButtonPartial.cs*

Once the file has been created, you'll see the following code when you open the file.

Code Listing 2-1: The Default Code of *TestButtonPartial.cs*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace BlazorApp.Components
{
    public class TestButtonPartial
    {
    }
}
```

To make this work, we need to make some changes. First, it is important that this code and the code from **TestButton.razor** are part of the same namespace, which in this case they are.

Both files were created in the same folder. Therefore, the namespace is the same for both: **BlazorApp.Components**.

The second thing we need to ensure is that the name of the class **TestButtonPartial** is renamed to **TestButton** and that the **partial** reserved word is added to the class declaration. So, then, the code would look as follows.

Code Listing 2-m: The Modified Code of *TestButtonPartial.cs*

```
namespace BlazorApp.Components
{
    public partial class TestButton
    {
        protected string
            TxtValue { get; set; } = "This is a partial class :)";
    }
}
```

Notice that the **using** statements have been removed because they are not required.

To make this example a bit more realistic and be able to appreciate how partial classes can be used, let's change the hard-coded text contained within the HTML markup of **TestButton.razor** and replace it with a property called **TxtValue**. The code with these changes follows, with the changes highlighted in bold.

Code Listing 2-n: Code Changed to *TestButton.razor*

```
<Button Class="custom-css"
        Style="margin: 10px;"
        Variant="@ButtonVariant.Contained"
        Color="@Color.Primary">
    @TxtValue
</Button>
```

Notice as well that the **@inherits** directive has been removed from **TestButton.razor**. **TestButton.razor** is no longer using the **TestButtonBase** inherited class contained within **TestButton.razor.cs** we previously created, and instead will use the partial **TestButton** class contained within **TestButtonPartial.cs**.

To test this, just rename **TestButton.razor.cs** to **_TestButton.razor.cs**, so you can be sure that **TestButton.razor** will not use the **TestButtonBase** class but the partial **TestButton** class from **TestButtonPartial.cs** instead.

Now, let's run the project by pressing **Ctrl+F5**. When navigating to the **Counter** page, we should see the **TestButton** component with the text that was assigned to the **TxtValue** property of the **TestButton** partial class contained within **TestButtonPartial.cs**.

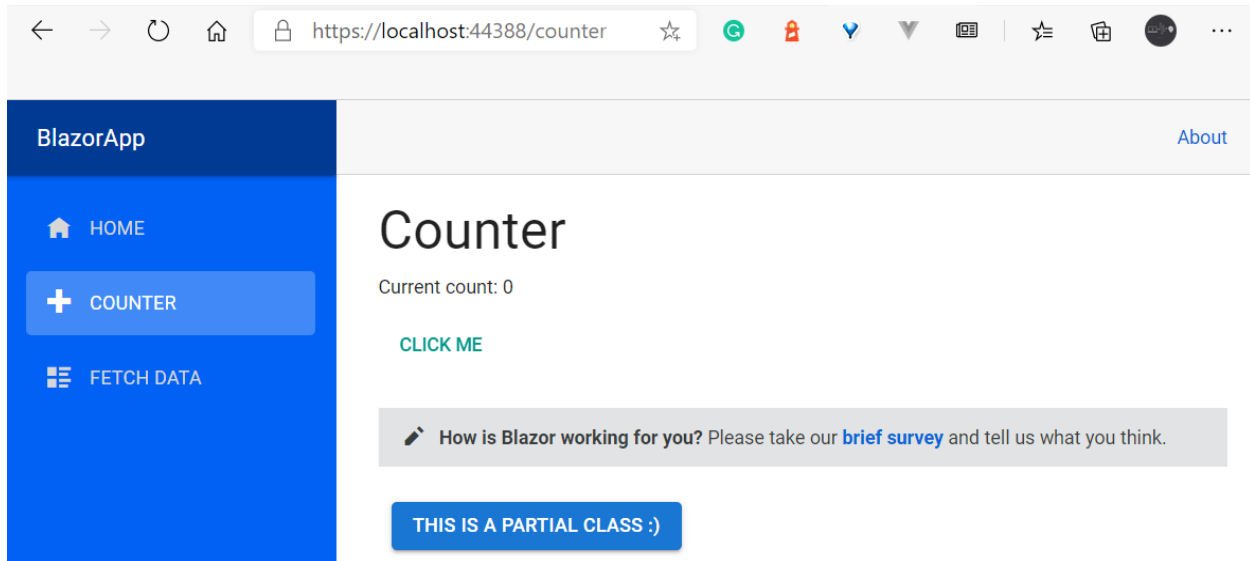


Figure 2-y: The TestButton Component (Using a Partial Class)

Switching back

Now that we have seen that it is possible to have the C# code of a component in a partial class, let's switch back to how things were.

So, open **Solution Explorer** and rename **TestButtonPartial.cs** to **_TestButtonPartial.cs**. Then, rename **_TestButton.razor.cs** back to **TestButton.razor.cs**.

Next, open **TestButton.razor.cs** and replace the existing code with the following.

Code Listing 2-o: Updated TestButton.razor.cs

```
using Microsoft.AspNetCore.Components;

namespace BlazorApp.Components
{
    public class TestButtonBase: ComponentBase
    {
        protected string
            TxtValue { get; set; } = "Using TestButtonBase :)";
    }
}
```

Furthermore, make sure that the updated code for the **TestButton.razor** file is as follows.

Code Listing 2-p: Updated TestButton.razor

```
@inherits TestButtonBase

<Button Class="custom-css"
        Style="margin: 10px;"
        Variant="@ButtonVariant.Contained"
        Color="@Color.Primary">
    @TxtValue
</Button>
```

Now, run the project by pressing **Ctrl+F5** and you should be able to see the following changes after navigating to the **Counter** page.

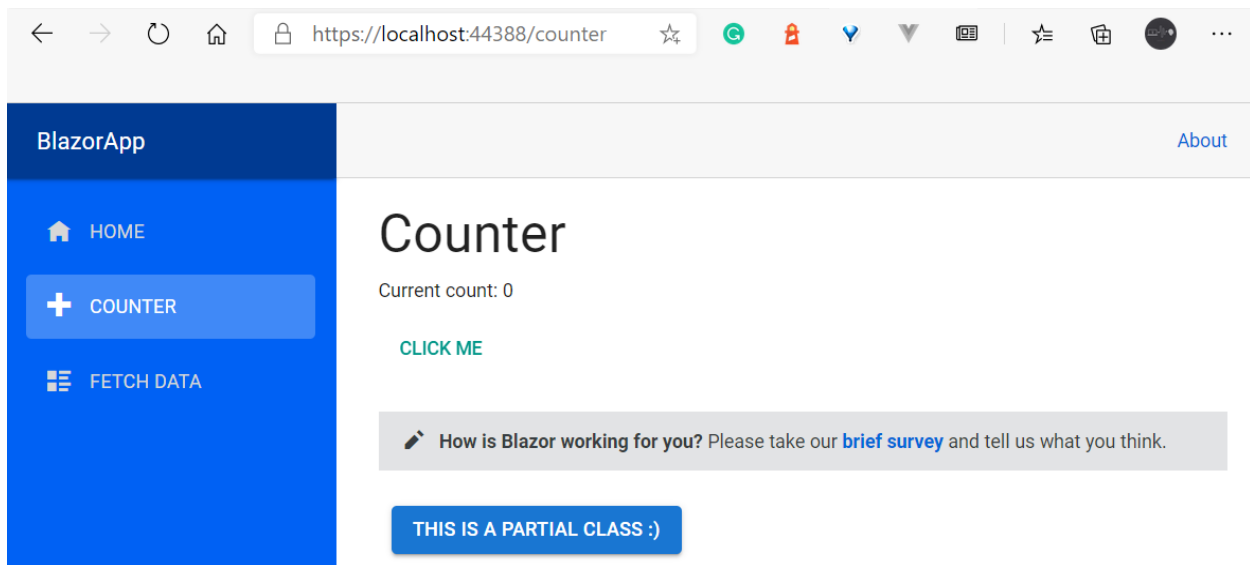


Figure 2-z: The TestButton Component (Still Using a Partial Class?)

So, why aren't we seeing a different result, even though **TestButtonPartial.cs** was renamed to **_TestButtonPartial.cs**, and **_TestButton.razor.cs** was renamed to **TestButton.razor.cs**?

The **TestButton** class contained within **TestButton.razor** references the code-behind partial class of **_TestButtonPartial.cs**, even though it inherits from the **TestButtonBase** class of **TestButton.razor.cs**.

To avoid this situation, simply change the name of the namespace of **_TestButtonPartial.cs** to **BlazorApp.Components.Partial** instead of **BlazorApp.Components**.

Once you make this code adjustment, you'll need to run the application again to see the changes. Notice that the hot reload feature won't pick this up, as it is not an HTML markup change.

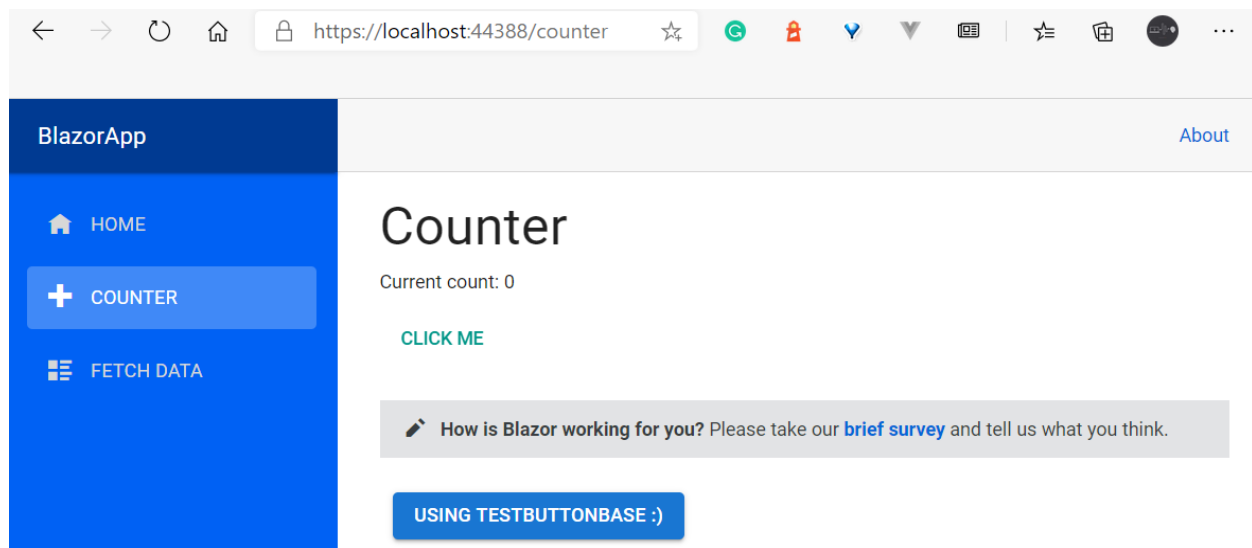


Figure 2-aa: The Updated UI (With the Expected Button Text)

One important lesson here is that even though partial classes are much easier to implement from a code perspective, they can lead to some confusion if a code-behind base class for the component already exists.

Therefore, the advice is to use one approach or the other, and avoid mixing them.

My personal preference is to use the base class code-behind approach we initially looked at, using file nesting, as it is easier to maintain even if it is a bit more work to set up.

Summary

We've covered quite a bit of ground throughout this chapter and started exploring what components are, how to reuse them, and also, most importantly, how to apply principles of separation of concerns between what is HTML markup and C# code.

Next, we will dive deeper into components and look at event handling, one-way data binding, and also how to place components into shared libraries.

Following that, we'll explore more advanced topics such as using the **Parameter** attribute, life cycle methods, two-way data binding, event callbacks, and other interesting advanced component features.

Chapter 3 Component Features

It's now time to dig a bit deeper into components and explore key component features such as event handling, one-way data binding, child content, and placing components into libraries for even further reusability.

This chapter is going to be a bit shorter than the previous one. However, all the topics that are going to be covered are not only exciting, they are essential features of Blazor to master. So, let's get started.

Event handling

To cope with events, each HTML element supports several Blazor event handlers available with a [delegate type](#) value.

All event handlers in Blazor start with the prefix **on**, which is what we will explore in this chapter.

To be able to start using and consuming events, go to **Solution Explorer** and open the **_Imports.razor** file. There, let's double-check whether we have the following **using** statement.

```
@using Microsoft.AspNetCore.Components.Web
```

By default, this reference should be present within the **_Imports.razor** file. Without it, we wouldn't be able to work with events, given that all event handlers are in that namespace.

So, to see this in action, let's go back to the **TestButton.razor** file. Let's add an **onclick** event to the button and also a **ColorValue** property, which is highlighted in bold in the following listing.

Code Listing 3-a: TestButton.razor (With the onclick Event)

```
@inherits TestButtonBase

<div @onclick="TestButtonClick">
    <Button Class="custom-css"
           Style="margin: 10px;"
           Variant="@ButtonVariant.Contained"
           Color="@ColorValue">
        @TxtValue
    </Button>
</div>
```

So, what have we done here? The first thing to notice is that we have placed the **onclick** event on a **div** element we previously didn't have, instead of placing it on the **Button** component. Why is that?

The reason is that the **Button** component, which is part of the Skclusive-UI library, is not a native HTML element, and because of that it doesn't implement the **onclick** event. Therefore, we have to wrap it around a native HTML element (like a **div**) that Blazor can use to execute an **onclick** event.

The other change is that we've added a **ColorValue** property. This property will be used to change the color of the **Button** component when it is clicked through the **div**.

For this to work, we also need to make some code changes to the **TestButton.razor.cs** code-behind file, which can be seen as follows. The changes are highlighted in bold.

Code Listing 3-b: Updated TestButton.razor.cs

```
using Microsoft.AspNetCore.Components;
using Skclusive.Core.Component;

namespace BlazorApp.Components
{
    public class TestButtonBase: ComponentBase
    {
        protected string
            TxtValue { get; set; } = "Using TestButtonBase :);";

        protected Color
            ColorValue { get; set; } = Color.Primary;

        protected void TestButtonClick()
        {
            ColorValue = (ColorValue == Color.Primary)
                ? Color.Secondary : Color.Primary;
        }
    }
}
```

We've added the **ColorValue** property to the **TestButton.razor.cs** code and given it a default value, which is the primary color.

Because the **ColorValue** property returns a **Color** object type, we have to import the namespace where this type resides, which in this case is **Skclusive.Core.Component**.

When the **onclick** event is triggered, the **TestButtonClick** method is executed. We can see that all this method does is swap the color from primary to secondary depending on which color is set. This is done using the [C# ternary operator](#).

If we now run the code by pressing **Ctrl+F5** and navigate to the **Counter** page, we should see the following.

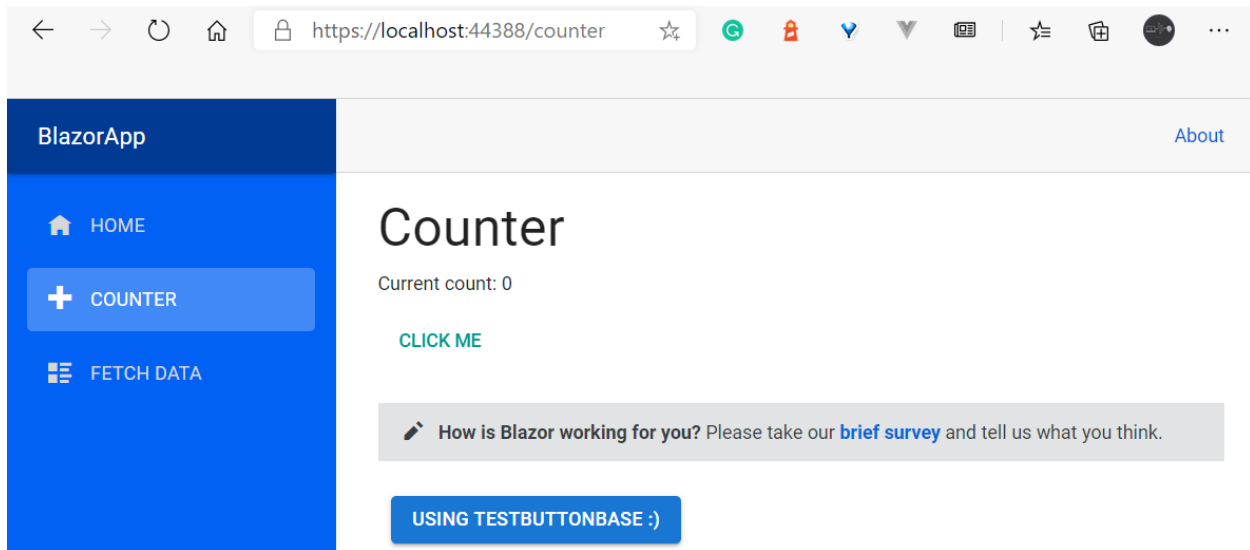


Figure 3-a: The TestButton Component (Counter Page)

If you click on the **TestButton** component, the button's color will change to the secondary color, as shown in the following figure.

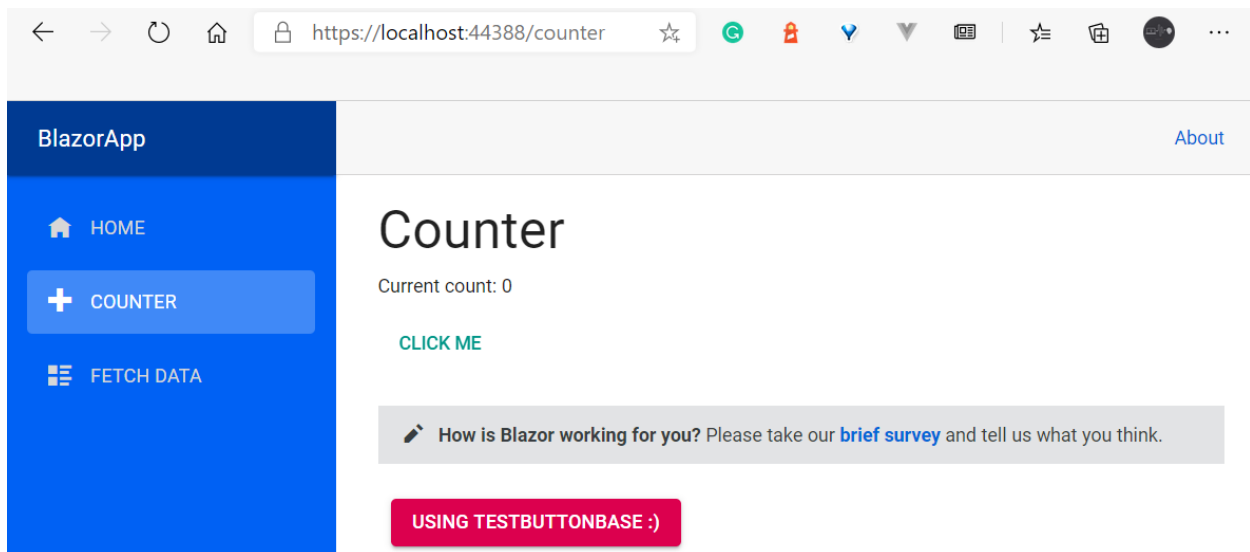


Figure 3-b: The TestButton Component with Color Changed (Counter Page)

If you click again on the **TestButton** component, the button's color will change back to the primary color.

So, by going through this example, we have implemented event handling.

One-way data binding

As we have just seen, when we click on the **TestButton**, the color of the button changes from the primary to the secondary color and vice versa, depending on the current color value.

This happens because the **ColorValue** property is assigned a value by the **TestButtonClick** method, and this property is data-bound to the **Color** attribute of the **Button** component.

The way this happens internally, using the Blazor server-side model, is that the server handles the changes between the server and the client using a SignalR connection.

When using the Blazor WebAssembly model, the change is completely handled in the browser by the WebAssembly runtime.

So, independently of the model being used, the process of updating these changes internally is handled in a very smart and efficient way, with virtually no network traffic.

To see how Blazor handles this internally, all you need to do is open the developer tools in your browser. On Windows, for [Chromium-based browsers](#) (I'm using [Microsoft Edge](#)), the developer tools can be opened by pressing **Ctrl+Shift+I**.

So, hover over the **TestButton** component on the **Counter** page UI, right-click, and then click on the **Inspect** menu option (or press **Ctrl+Shift+I**).

Now, you'll be able to see the actual HTML that Blazor has rendered. The **TestButton** component's rendered HTML has been highlighted in the following figure.

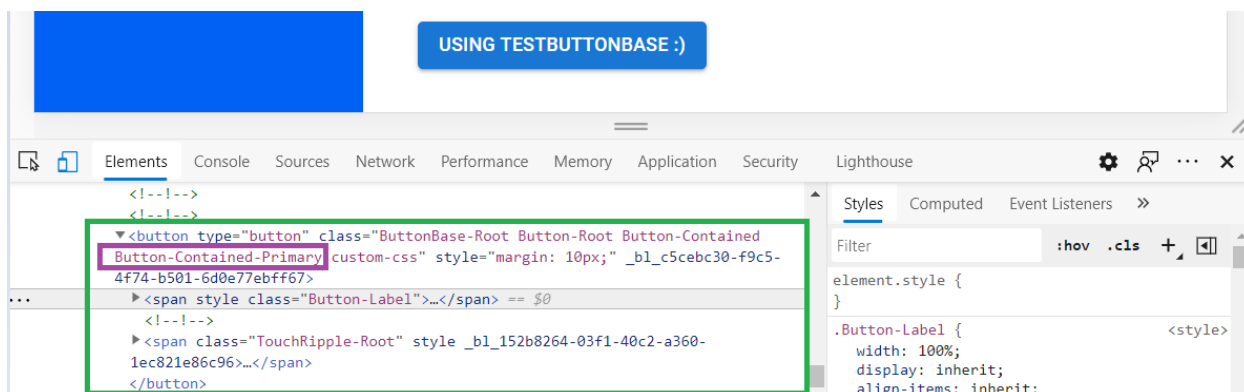


Figure 3-c: Inspecting the TestButton Component, Primary Color (Developer Tools)

Notice that Blazor has applied the CSS class **Button-Contained-Primary** when the primary color is being used.

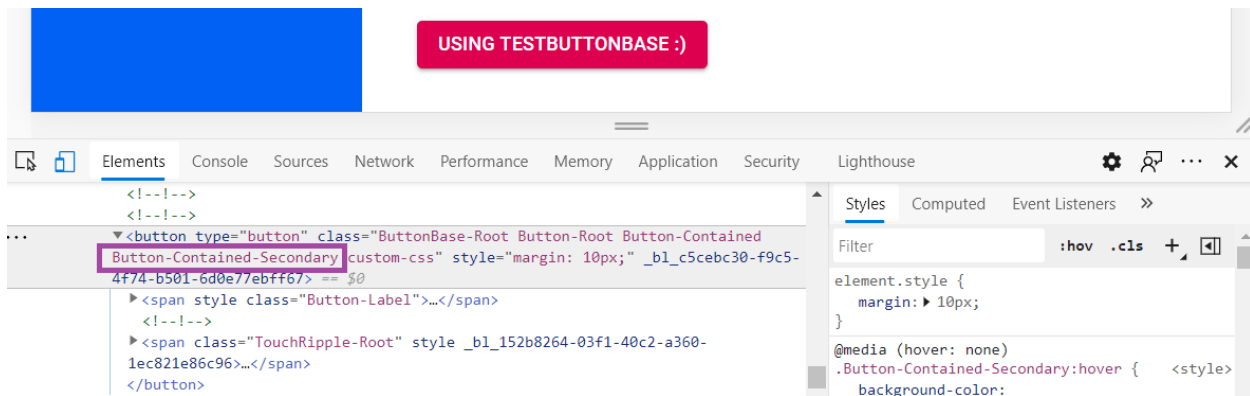


Figure 3-d: Inspecting the TestButton Component, Secondary Color (Developer Tools)

When the **TestButton** component is assigned the secondary color, Blazor updates the CSS class to **Button-Contained-Secondary**, as can be seen in the preceding figure.

If you paid close attention when clicking the button, you saw the browser developer tools highlighted with a color the part of the HTML that changed in real time.

That small change to the HTML is possible because of an internal feature that exists in Blazor called the diff mechanism.

The way the diff mechanism works is that Blazor quickly builds a new page with the new details, without rendering it, and compares it to the old one. Only the differences between the pages will be applied to the existing rendered page.

Because of this, UI updates executed by Blazor are very efficient as the browser doesn't have to build the page every time there is a change. [The diff is sent to the client in a binary format and applied by the browser.](#)

This mechanism even works when navigating to a different page. So, the diff mechanism goes hand-in-hand with one-way data binding.

Child content

To demonstrate how child content can be passed to a component, we are going to use a feature of Blazor called **RenderFragment**.

Let's say we want to add some text to the **TestButton** component, just below the actual **Button** component, but without adding any additional child components to the **TestButton** component.

To do that, we need a public property of type **RenderFragment** called **ChildContent**. In this case, the name of the property matters, so it has to be named exactly, **ChildContent**. If another name is given, it won't work.

In Blazor, when a property needs to be set by another component, it needs to be decorated with the **Parameter** attribute.

So, let's open the **TestButton.razor.cs** file using **Solution Explorer** and make these adjustments. The changes are highlighted in bold in the following listing.

Code Listing 3-c: Updated TestButton.razor.cs

```
using Microsoft.AspNetCore.Components;
using Skclusive.Core.Component;

namespace BlazorApp.Components
{
    public class TestButtonBase: ComponentBase
    {
        protected string
            TxtValue { get; set; } = "Using TestButtonBase :)";

        protected Color
            ColorValue { get; set; } = Color.Primary;

        protected void TestButtonClick()
        {
            ColorValue = (ColorValue == Color.Primary)
                ? Color.Secondary : Color.Primary;
        }

        [Parameter]
        public RenderFragment ChildContent { get; set; }
    }
}
```

Next, we need to open the **TestButton.razor** file using **Solution Explorer** and add the **ChildContent** within a **div** element.

We can see this as follows. The changes are highlighted in bold.

Code Listing 3-d: Updated TestButton.razor

```
@inherits TestButtonBase

<div @onclick="TestButtonClick">
    <Button Class="custom-css"
        Style="margin: 10px;"
        Variant="@ButtonVariant.Contained"
        Color="@ColorValue">
        @TxtValue
    </Button>
    <br />
</div>
```

```
<div>@ChildContent</div>
</div>
```

Next, we need to open the **Counter.razor** file (which resides in the **Pages** folder) using **Solution Explorer**, and there make the following adjustments, which are highlighted in the following code listing in bold.

Code Listing 3-e: Updated Counter.razor (Using ChildContent)

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

<SurveyPrompt Title="How is Blazor working for you?" />
<TestButton>
    This is the child content
</TestButton>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

As you can see, the text is passed to the **TestButton** component as a **ChildContent**, between the **<TestButton>** and **</TestButton>** tags.

Class libraries and component sharing

The ability to share as well as the reusability of a component can be expanded even further by placing components that are likely to be used in other projects into a component library—which is nothing other than a standard Razor class library.

Creating a new component in a class library is very simple. We can do this by going to **Solution Explorer** and right-clicking on the solution name (the top element in **Solution Explorer**), and then clicking **Add**, and after that clicking **New Project**.

Then, in the search box, type **Razor Class** and choose the **Razor Class Library** project template, as shown in the following figure.

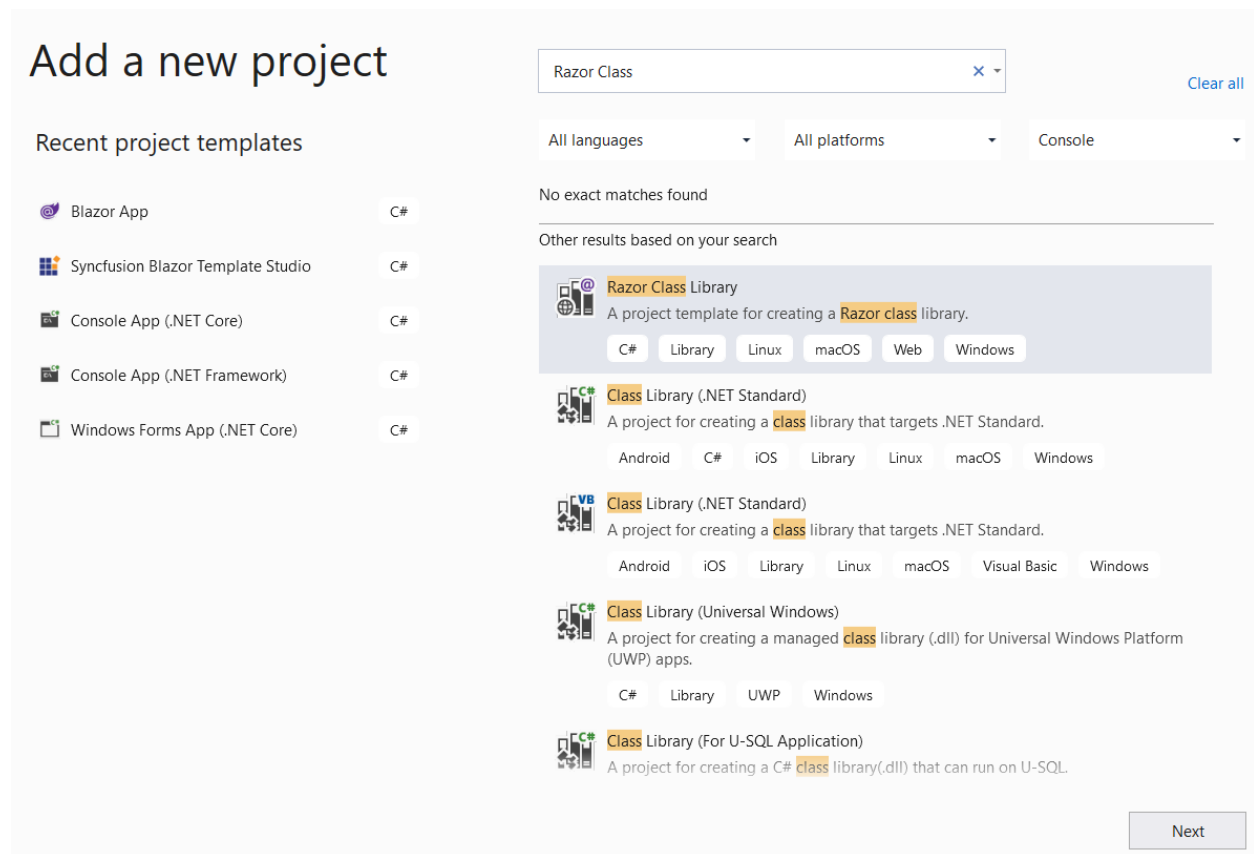


Figure 3-e: Adding a New Razor Class Library

Give it a name—let's call it **ComponentLib**—and then click the **Create** button. Once the class library has been created, it will be shown as a project within **Solution Explorer**, as part of the **BlazorApp** solution.

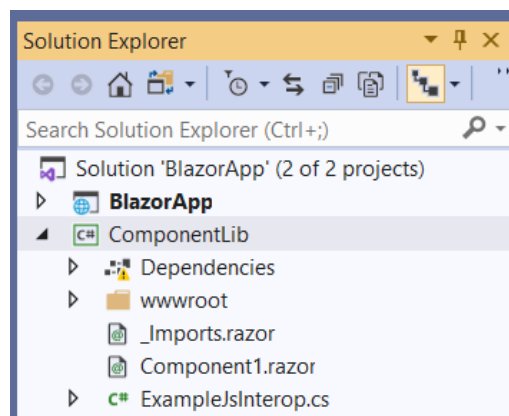


Figure 3-f: The ComponentLib Razor Class Library

To be able to use the **ComponentLib** library from the main **BlazorApp** project, we need to add it as a dependency.

This can be done using **Solution Explorer**, selecting the **BlazorApp** project (not the **BlazorApp** solution), right-clicking, clicking the option **Add Project Reference**, and then selecting the **ComponentLib** item from the list, as can be seen in the following figure.

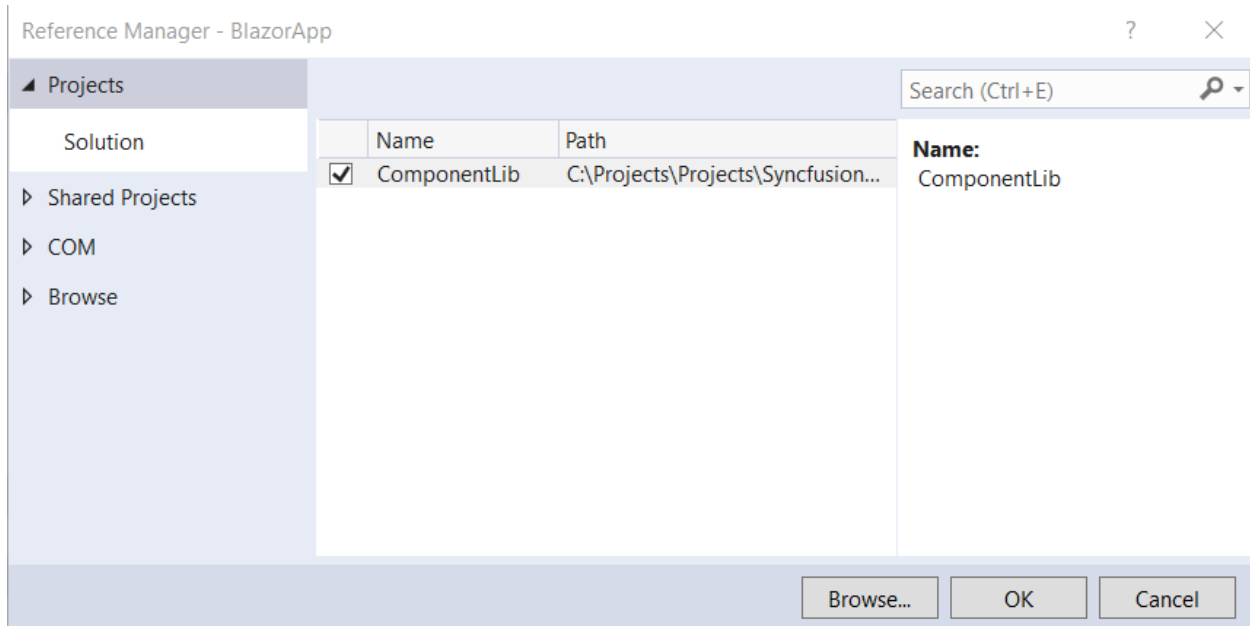


Figure 3-g: Adding ComponentLib to the BlazorApp Project

Once selected, click **OK**. By doing this, the **ComponentLib** library will now be accessible to the **BlazorApp** project.

However, before we can use any component defined within the **ComponentLib** library, we need to add the namespace to the **_Imports.razor** file of the **BlazorApp** project. We can do that by adding the following line.

@using ComponentLib

So, to test this, in **Solution Explorer**, open the **Counter.razor** file, which resides in the **Pages** folder, and let's add the following line (highlighted in bold) to the existing code.

Code Listing 3-f: Updated Counter.razor (Using a ComponentLib Component)

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

```

<SurveyPrompt Title="How is Blazor working for you?" />
<TestButton>
    This is the child content
</TestButton>

<Component1 />

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}

```

What we have done here is add the **Component1** component (which can be found within the **Component1.razor** file under the root folder of the **ComponentLib** project) to the source code of the **Counter.razor** file.

If we now run the application by pressing **Ctrl+F5**, and then navigate to the **Counter** page, we can see the following.

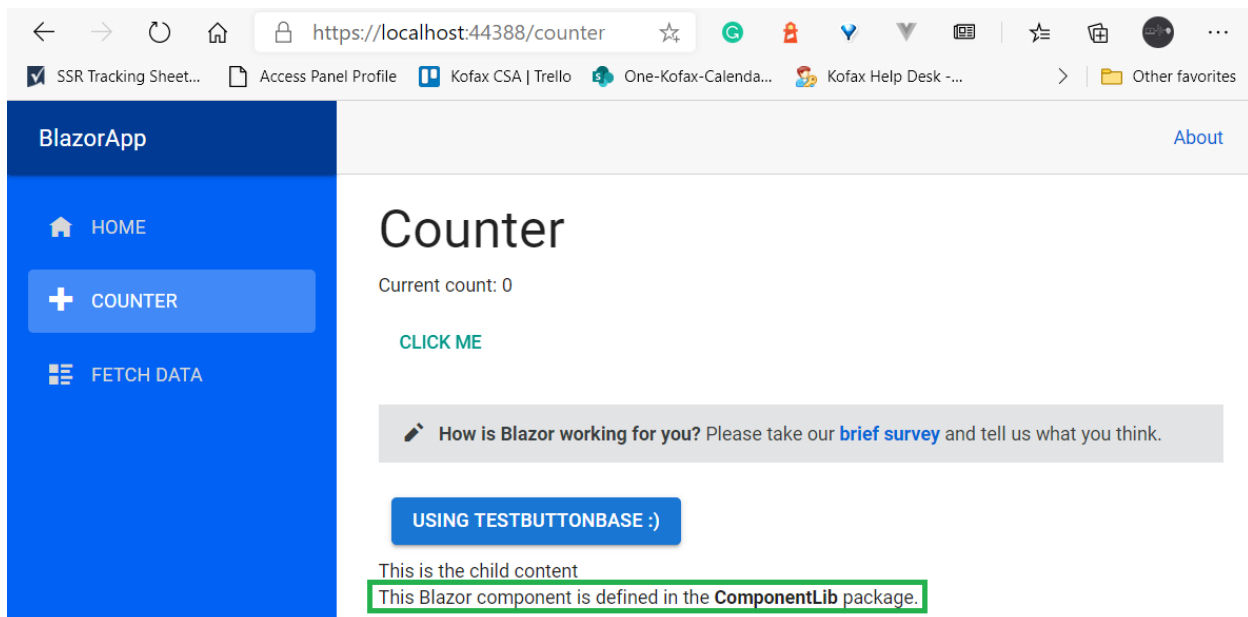


Figure 3-h: The Counter Page with Component1 Embedded

As you can see, **Component1** from the **ComponentLib** library has been embedded within the **Counter** page.

So, using a library is a great way to share components among various projects. This is particularly useful when you have a set of components that you want to share between a Blazor Server app and a Blazor WebAssembly project.

Summary

Throughout this chapter, we explored some fundamental aspects of Razor components. Without some good knowledge of these, it would be very difficult to build any Blazor application.

Next, we'll explore more advanced component features that will allow us to compose a Blazor application even further.

These features include topics such as object injection, component life cycle methods, how to preserve elements, conditional rendering, two-way data binding, using event callbacks, and referencing components.

So, we have quite a bit of ground to cover and a lot of information and techniques to digest—all very exciting things ahead.

Chapter 4 Using Components

So far, we've covered the basics of working with components in Blazor, but we have yet to explore other features that will allow us to use components and add additional features to a Blazor application. That's what this chapter is all about.

Conditional rendering

One of the great things about working with frameworks like Blazor is that there's the possibility to show or not show parts of the UI based on specific conditions.

On the **Weather forecast** page, let's add a new feature for each row, where an additional section with extra information can be shown for each data row when an arrow is clicked.

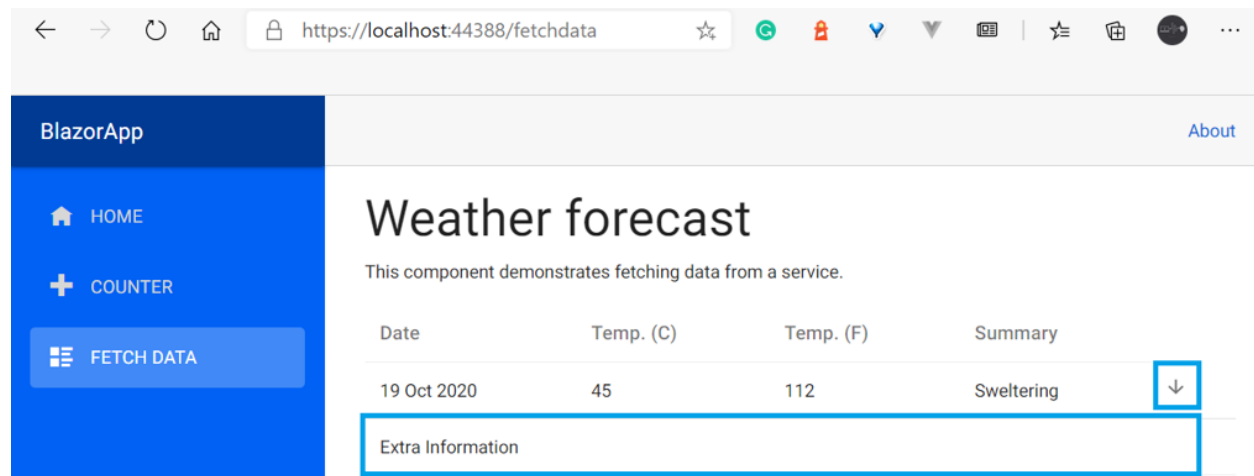


Figure 4-a: Weather Forecast with Additional Details (Which We Have Yet to Create)

So, to be able to make this work, we need to do a few things. Let's start by exploring the current code that displays row data. This is available within the **FetchData.razor** page, which resides in the **Pages** folder. Let's open this file using **Solution Explorer**.

Code Listing 4-a: FetchData.razor

```
@page "/fetchdata"

@using BlazorApp.Data
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>
```

```

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }
}

```

From the preceding listing, the HTML markup responsible for rendering each row of data has been highlighted in bold.

To add the additional section, the first thing we need to do is refactor this code into a separate component. We are going to call it **ForecastRow.razor** and place it into the **Components** folder.

So, using **Solution Explorer**, right-click on the **Components** folder within the **BlazorApp** project, click **Add**, and then click the **Razor Component** option. Then, name the component **ForecastRow.razor** and click **Add**.

Now, open the **ForecastRow.razor** file and add the following code.

Code Listing 4-b: ForecastRow.razor

```
@inherits ForecastRowBase

<tr>
    <td>@Forecast.Date.ToShortDateString()</td>
    <td>@Forecast.TemperatureC</td>
    <td>@Forecast.TemperatureF</td>
    <td>@Forecast.Summary</td>
    <div @onclick="e => Forecast.ShowExtras = !Forecast.ShowExtras">
        <IconButton aria-label="delete"
            Style="margin: 8px; "
            Size="@IconButtonSize.Small">
            <ArrowDownwardIcon Style="font-size: inherit; " />
        </IconButton>
    </div>
</tr>
@if (Forecast.ShowExtras)
{
    <tr>
        <td colspan="5">
            Extra Information
        </td>
    </tr>
}
}
```

If you get a warning that “**Element ‘div’ cannot be nested inside element ‘tr,’**” you can avoid this warning by placing the **<div>** inside a **<td>** cell.

The first thing to notice is that the **ForecastRow** component inherits from the **ForecastRowBase** class.

This means that the code-behind for the **ForecastRow.razor** file will reside in a file we have yet to create called **ForecastRow.razor.cs**, which will contain the **ForecastRowBase** class. We’ll do this in a bit.

The next thing to notice is that we’ve taken each row item and changed the references from **forecast** to **Forecast**. This might seem like a minor change, but it is quite significant.

By making this change, we are saying that the **ForecastRow** component is no longer getting the forecast data through the **forecast** object, which is an instance of the **WeatherForecast** class declared within the **WeatherForecast.cs** file under the **Data** folder.

Instead, the **ForecastRow** component is getting the forecast data through the **Forecast** property. It will be of type **WeatherForecast**, and is going to be part of the **ForecastRowBase** class that the **ForecastRow** component inherits from. We'll come back to this point shortly.

Next, we also notice that we've added a **div** that contains an **IconButton**, which is the arrow highlighted in the preceding figure.

Just like we did with the **TestButton** component, the **onclick** event has been added to the **div**, which is a real HTML element, and not to the **IconButton**.

The **onclick** event contains an inline [C# lambda operator](#), which is described as follows.

```
e => Forecast.ShowExtras = !Forecast.ShowExtras
```

This toggles (shows or hides) the additional information section, which comes after the **@if (Forecast.ShowExtras)** instruction.

If the value of the **Forecast.ShowExtras** property is set to **true**, the panel is shown. On the contrary, if the value of **Forecast.ShowExtras** is set to **false**, the panel is hidden.

Notice as well that the panel spans the length of the five columns of the data row, which is achieved with the **<td colspan="5">** instruction.

Now that we've explored the **ForecastRow** component, let's create the **ForecastRow.razor.cs** file, which is the code-behind file for the component.

So, using **Solution Explorer**, click on the **Components** folder, right-click, click **Add**, click **New Item**, and then select the **Class** item from the list. Let's give it the name **ForecastRow.razor.cs**, and then click **Add**.

Once created, add the following code to **ForecastRow.razor.cs**.

Code Listing 4-c: ForecastRow.razor.cs

```
using BlazorApp.Data;
using Microsoft.AspNetCore.Components;

namespace BlazorApp.Components
{
    public class ForecastRowBase: ComponentBase
    {
        [Parameter]
        public WeatherForecast Forecast { get; set; }
    }
}
```

There are two things to notice. The first is that we've added a reference to the **BlazorApp.Data** namespace. This is because the **Forecast** property is of type **WeatherForecast**.

The second thing to notice is that the **Forecast** property is decorated with the **Parameter** attribute. This means that an instance of **WeatherForecast** will be passed from the parent to the child component, **ForecastRow**.

To invoke the **ForecastRow** component, we need to modify the **FetchData.razor** file. We'll do this shortly.

But before we do that, there's one crucial piece to this conditional rendering puzzle we still need to complete, and that is to add the **ShowExtras** property to the **WeatherForecast** class.

So, using **Solution Explorer**, open the **WeatherForecast.cs** file that resides in the **Data** folder, and let's add the **ShowExtras** property.

The updated code for the **WeatherForecast** class follows. The change is highlighted in bold.

Code Listing 4-d: Updated WeatherForecast.cs

```
using System;

namespace BlazorApp.Data
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }

        public int TemperatureC { get; set; }

        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

        public string Summary { get; set; }

        public bool ShowExtras { get; set; }
    }
}
```

Invoking ForecastRow

Let's invoke the **ForecastRow** component we've just created. Using **Solution Explorer**, navigate to the **Pages** folder and open the **FetchData.razor** file. Once the file is open, select the following code.

```
<tr>
    <td>@forecast.Date.ToShortDateString()</td>
```

```

        <td>@forecast.TemperatureC</td>
        <td>@forecast.TemperatureF</td>
        <td>@forecast.Summary</td>
    </tr>

```

Replace this code with `<ForecastRow Forecast="forecast" />`.

After making this change, the complete code for the **FetchData.razor** page should be as follows.

Code Listing 4-e: Updated FetchData.razor

```

@page "/fetchdata"

@using BlazorApp.Data
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

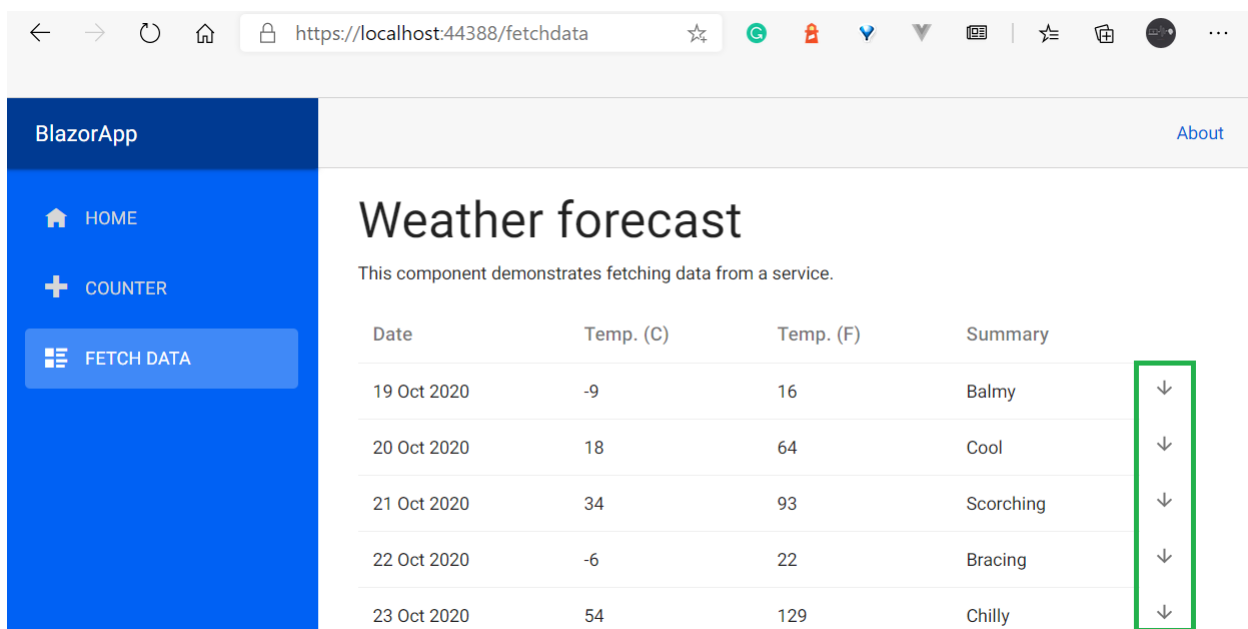
@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <ForecastRow Forecast="forecast" />
            }
        </tbody>
    </table>
}

```

```
@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }
}
```

If we now run the application by pressing **Ctrl+F5**, we will see the following when navigating to the **Weather forecast** page, which we can do by clicking the **FETCH DATA** option on the app's menu.



The screenshot shows a web browser at <https://localhost:44388/fetchdata>. The application is titled "BlazorApp" and has a sidebar with three options: "HOME", "COUNTER", and "FETCH DATA" (which is selected). The main content area is titled "Weather forecast" and includes a description: "This component demonstrates fetching data from a service." Below this is a table with the following data:

Date	Temp. (C)	Temp. (F)	Summary
19 Oct 2020	-9	16	Balmy
20 Oct 2020	18	64	Cool
21 Oct 2020	34	93	Scorching
22 Oct 2020	-6	22	Bracing
23 Oct 2020	54	129	Chilly

Each row in the table has a downward arrow icon on the right side, which is highlighted with a green box in the image.

Figure 4-b: Updated Weather forecast Page

If you click on any of the arrows, the extras panel will appear, and if you click again on the same arrow, the panel will hide.

At this stage, we have added the basic functionality of the panel, which is part of the **ForecastRow** component.

However, we need to add some extra features to this component, such as the ability to display wind and humidity data when the panel is shown, for each row of data. That's what we'll do next.

Object injection and initialization

To be able to complete the functionality of the **ForecastRow** component, we inject an object that will contain wind and humidity information for each row.

So, let's open the **ForecastRow.razor.cs** file that resides in the **Components** folder using **Solution Explorer**. Then we'll make some changes to the existing code, which are highlighted in bold in the following listing.

Code Listing 4-f: Updated Forecast.razor.cs

```
using BlazorApp.Data;
using Microsoft.AspNetCore.Components;
using System.Threading.Tasks;

namespace BlazorApp.Components
{
    public class ForecastRowBase: ComponentBase
    {
        [Parameter]
        public WeatherForecast Forecast { get; set; }

        protected WeatherExtras Extras { get; set; }

        [Inject]
        public WeatherExtrasService ExtrasService { get; set; }

        protected async override Task OnInitializedAsync()
        {
            Extras = await ExtrasService.GetExtrasAsync(Forecast);
        }
    }
}
```

The first thing to notice is that we've added to the **Forecast.razor.cs** code a reference to the **System.Threading.Tasks** namespace. This is because the **OnInitializedAsync** life cycle event returns a **Task** object.

Next, we've added an **Extras** property of type **WeatherExtras** (which is a class we have not created yet). The **Extras** property will contain the wind and humidity information.

Then, we have the **ExtrasService** property (which has not been created yet) injected into the **ForecastRowBase** class. This is why the property is decorated with the **Inject** attribute.

ExtrasService will be used to retrieve the additional weather information that will be assigned to the **Extras** property.

The retrieval of the information will be done on the **OnInitializedAsync** event, which is a method that overrides the **base** method with the same name that gets inherited from the **ComponentBase** class. This is one of the various component life cycle events.

Within **OnInitializedAsync**, the **GetExtrasAsync** method from the **WeatherExtrasService** instance is invoked, which returns a **WeatherExtras** type object.

The current weather forecast (the **Forecast** object, which contains the data row details) is passed as a parameter to the **GetExtrasAsync** method.

Extras object model

Now, let's create the **WeatherExtras** class. To do that, let's create a new file called **WeatherExtras.cs** under the **Data** folder.

Using **Solution Explorer**, click on the **Data** folder, right-click, click **Add**, click **New Item**, and select **Class** from the items list. Name the file **WeatherExtras.cs**, and then click **Add**.

Once the file has been created, open it, and let's add the following code to it.

Code Listing 4-g: WeatherExtras.cs

```
namespace BlazorApp.Data
{
    public class WeatherExtras
    {
        public WeatherForecast Forecast { get; set; }

        public int Humidity { get; set; }

        public string Wind { get; set; }
    }
}
```

As you can see, there's nothing too complicated about this code. We are simply declaring the **Forecast**, **Humidity**, and **Wind** properties.

This class represents the model for the extra weather data—nothing else. That was easy.

Extras service

Now let's shift our attention to the service (**WeatherExtrasService**) that will be responsible for retrieving the additional weather data and assigning it to the model.

Using **Solution Explorer**, right-click the **Data** folder, then click **Add**, and after that click **New Item** and select **Class** from the items list. Name the file **WeatherExtrasService.cs**, and then click **Add**.

Once the file has been created, open it, and let's add the following code to it.

Code Listing 4-h: WeatherExtrasService.cs

```
using System;
using System.Threading.Tasks;

namespace BlazorApp.Data
{
    public class WeatherExtrasService
    {
        private static readonly string[] WindTypes = new[]
        {
            "Strong", "Heavy", "Mild", "Chilly", "No wind",
            "Hurricane", "Bursts", "Gentle breeze"
        };

        public Task<WeatherExtras> GetExtrasAsync(WeatherForecast forecast)
        {
            var rng = new Random();
            return Task.FromResult(new WeatherExtras
            {
                Forecast = forecast,
                Humidity = rng.Next(-20, 55),
                Wind = WindTypes[rng.Next(WindTypes.Length)]
            });
        }
    }
}
```

The code for this service is quite similar to the code of the **WeatherForecastService.cs** file that came out of the box when the project was created.

The **GetExtrasAsync** method simply creates a new instance of the **WeatherExtras** class and assigns random data to the **Humidity** and **Wind** properties of the object created, which is then returned by the method.

The value of the **Humidity** property is assigned based on random numbers. The value of the **Wind** property is assigned based on a random choice of any of the items contained within the **WindTypes** array.

In a real-world application, a service like this would be responsible for retrieving the data from the back end, such as a SQL Server instance. In our case, we are just using randomly generated data.

Registering the service

Before the service can be used, it needs to be registered in the **ConfigureServices** method of the **Startup.cs** file that resides in the project's root folder. So, let's do that.

Using **Solution Explorer**, open the **Startup.cs** file and locate the **ConfigureServices** method. Then add the following line.

```
services.AddSingleton<WeatherExtrasService>();
```

The updated **ConfigureServices** method should look as follows.

Code Listing 4-i: Registering WeatherExtrasService (ConfigureServices Method of Startup.cs)

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddServerSideBlazor();
    services.AddSingleton<WeatherForecastService>();

    services.AddSingleton<WeatherExtrasService>();

    services.TryAddMaterialServices(new
        MaterialConfigBuilder().Build());
}
```

Updating ForecastRow.razor

Before we can try this new functionality, I'd like to make a small *cosmetic* update to the HTML markup of the **ForecastRow** component so that the **Humidity** and **Wind** values can be visible.

In the listing that follows, I've highlighted the changes in bold.

Code Listing 4-j: Updated ForecastRow.razor File

```
@inherits ForecastRowBase

<tr>
    <td>@Forecast.Date.ToShortDateString()</td>
    <td>@Forecast.TemperatureC</td>
```

```

<td>@Forecast.TemperatureF</td>
<td>@Forecast.Summary</td>
<div @onclick="e => Forecast.ShowExtras = !Forecast.ShowExtras">
  <IconButton aria-label="delete"
    Style="margin: 8px; "
    Size="@IconButtonSize.Small">
    <ArrowDownwardIcon Style="font-size: inherit; " />
  </IconButton>
</div>
</tr>
@if (Forecast.ShowExtras)
{
<tr>
  <td>
    <strong>Humidity:</strong> @Extras.Humidity%
  </td>
  <td>
    <strong>Wind:</strong> @Extras.Wind
  </td>
</tr>
}

```

All I've done is render the values of the **Humidity** and **Wind** properties from the **Extras** instance.

If we now run the application by pressing **Ctrl+F5**, we'll be able to see the changes when navigating to the **Weather forecast** page. Just click on a couple of arrows on different rows to see the additional information.

20 Oct 2020	29	84	Cool	↓
<div> <div>Humidity: -7%</div> <div>Wind: No wind</div> </div>				
21 Oct 2020	-8	18	Cool	↓
22 Oct 2020	-11	13	Hot	↓
<div> <div>Humidity: -15%</div> <div>Wind: Chilly</div> </div>				
23 Oct 2020	37	98	Cool	↓

Figure 4-c: Updated Weather Forecast with Extras

Because the data is generated randomly, don't expect the values to make much sense. The important takeaway is how to compose and use components within a Blazor application.

Life cycle methods

We just saw with the `WeatherExtrasService` class that by using the `OnInitializedAsync` method, we were able to retrieve the data to assign to the `Extras` property of the `ForecastRowBase` instance that the `ForecastRow` component inherits from.

Of all the life cycle methods Blazor components can use, you'll probably be using `OnInitializedAsync` the most. Nevertheless, there are other life cycle methods available. Let's have a look at some of them.

There's the `OnParametersSetAsync` method (the synchronous version of this method is called `OnParametersSet`), which is triggered each time a component receives a new parameter from its parent.

If you'd like to implement this method in your component, it would have to be as follows.

Code Listing 4-k: Implementing `OnParametersSetAsync`

```
protected override Task OnParametersSetAsync()
{
    return base.OnParametersSetAsync();
}
```

This method could be used, for instance, to set a private field to a value that is dependent on one of the incoming properties.

There's also the `SetParametersAsync` method (the synchronous version of this method is called `SetParameters`), which can be used if you would like to do something before the parameters passed to the component are assigned to the properties.

Code Listing 4-l: Implementing `SetParametersAsync`

```
public override Task SetParametersAsync(ParameterView parameters)
{
    return base.SetParametersAsync(parameters);
}
```

Notice that this method must be **public** rather than **protected**. The incoming parameters can be inspected by looking at the `parameters` argument, which is of `ParameterView` type.

It's important to know that the assignment of the incoming parameters to the properties of the component is done by the `base` implementation of the method.

This means that if the call to the **base** implementation of the method is omitted, the incoming parameters won't be assigned to the component properties.

There's also the **OnAfterRenderAsync** (the synchronous version of this method is called **OnAfterRender**). When this method gets triggered, the component has already been re-rendered.

This method also fires the first time the component renders, so in that case the **firstRender** argument passed to this method has a value of **true**.

Code Listing 4-m: Implementing OnAfterRenderAsync

```
protected override Task OnAfterRenderAsync(bool firstRender)
{
    return base.OnAfterRenderAsync(firstRender);
}
```

You can use this method if you would like to do something with the rendered elements, such as using a specific JavaScript library that uses them.

Then, we also have the **ShouldRender** method. If **false** is returned from this method, it can prevent the component from re-rendering.

Code Listing 4-n: Implementing ShouldRender

```
protected override bool ShouldRender()
{
    return base.ShouldRender();
}
```

Blazor ignores this method when a component is initially rendered.

Element preservation

One important thing to keep in mind when working with life cycle methods is that Blazor's diff mechanism is in charge, so it will always try to update the element that changed within a list of elements.

Blazor's diff mechanism does its best to map objects and elements that have changed on its own, but the way it does this is not perfect.

This means that sometimes the diff mechanism might lose track of what changes have taken place and it might not re-render an element that changed, but instead re-render the whole list of elements the changed element is part of.

When situations like these occur, they could lead to unexpected side effects and affect the user experience with the application.

This not only applies to changes of a specific element within a list, but also to operations such as insertion or deletion of elements in a list.

In such cases, when something goes wrong, the diff mechanism will have no idea where to place the new element.

So, to prevent these types of situations, we as developers can provide some help to the diff mechanism in the way we write our code.

The way we can do this is by adding a key to each element in the list. The key should be unique for each element. So, it can be a unique identifier that doesn't repeat itself, or it can be an object reference.

The diff mechanism works not only by comparing unique values for keys but also by comparing different object references. Let's go ahead and implement this feature to see how simple it is.

Using **Solution Explorer**, open the **FetchData.razor** file that resides in the **Pages** folder. There, find the reference to the **ForecastRow** object.

```
<ForecastRow Forecast="forecast" />
```

Let's change it to this.

```
<ForecastRow @key="forecast" Forecast="forecast" />
```

That's all—if you now run the application with **Ctrl+F5** and navigate to the **Weather forecast** page, you will see that everything works as usual.

Two-way data binding

To understand the simplicity and power of two-way data binding, let's add an input check box to the extra details section. This check box will have the same value as **Forecast.ShowExtras**.

So, when the arrow button is clicked, the extra details section for that row will be shown, and the check box will be selected. When the check box is cleared, the row will be hidden.

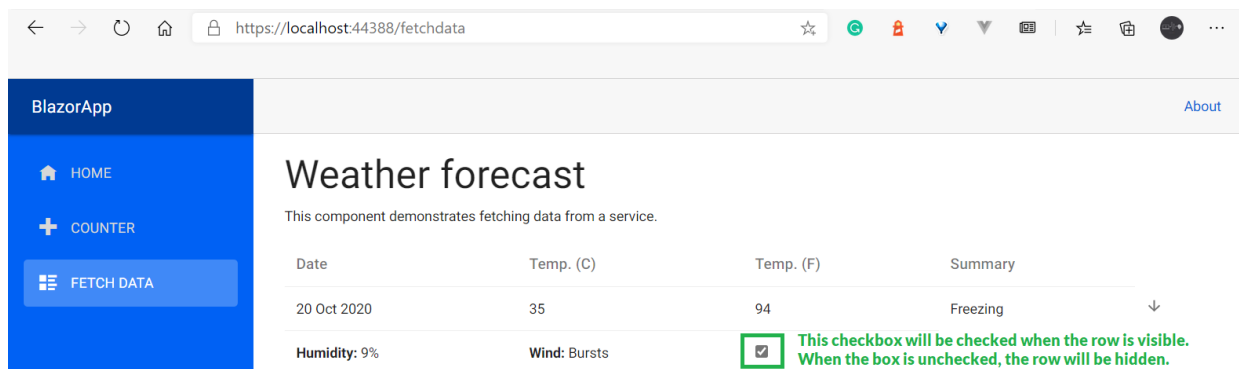


Figure 4-d: Extras Section with Check Box Bound to `Forecast.ShowExtras`

So, how can we achieve this with what we already know? Using **Solution Explorer**, go to the **Components** folder and open the **ForecastRow.razor** file.

Let's update the existing code with the following. The changes are highlighted in bold.

Code Listing 4-o: Updated ForecastRow.razor

```
@inherits ForecastRowBase

<tr>
    <td>@Forecast.Date.ToShortDateString()</td>
    <td>@Forecast.TemperatureC</td>
    <td>@Forecast.TemperatureF</td>
    <td>@Forecast.Summary</td>
    <div @onclick="e => Forecast.ShowExtras = !Forecast.ShowExtras">
        <IconButton aria-label="delete"
            Style="margin: 8px;"
            Size="@IconButtonSize.Small">
            <ArrowDownwardIcon Style="font-size: inherit;" />
        </IconButton>
    </div>
</tr>
@if (Forecast.ShowExtras)
{
    <tr>
        <td>
            <strong>Humidity:</strong> @Extras.Humidity%
        </td>
        <td>
            <strong>Wind:</strong> @Extras.Wind
        </td>
        <td>
            <input type="checkbox" checked="@Forecast.ShowExtras"
                @onchange="e => base.CheckboxChanged(e)"/>
        </td>
    </tr>
}
```

As we can see from the preceding listing, we've added another element to the extras section, an input check box, bound to **Forecast.ShowExtras**, which is set when the arrow button is clicked.

The check box has also an **onchange** event that is triggered every time the check box is clicked. This invokes the **CheckboxChanged** method from the **base** class of the **ForecastRow** component, which is **ForecastRowBase**.

Next, let's add the **CheckboxChanged** method to the **ForecastRow.razor.cs** file. The updated code for the **ForecastRowBase** class follows, and the changes are highlighted in bold.

Code Listing 4-p: Updated ForecastRow.razor.cs

```
using BlazorApp.Data;
using Microsoft.AspNetCore.Components;
using System.Threading.Tasks;

namespace BlazorApp.Components
{
    public class ForecastRowBase: ComponentBase
    {
        [Parameter]
        public WeatherForecast Forecast { get; set; }

        protected WeatherExtras Extras { get; set; }

        [Inject]
        public WeatherExtrasService ExtrasService { get; set; }

        protected async override Task OnInitializedAsync()
        {
            Extras = await ExtrasService.GetExtrasAsync(Forecast);
        }

        public void CheckboxChanged(ChangeEventArgs e)
        {
            var n = (bool)e.Value;
            Forecast.ShowExtras = n;
        }

        protected override Task OnParametersSetAsync()
        {
            return base.OnParametersSetAsync();
        }

        public override Task SetParametersAsync(ParameterView parameters)
        {
            return base.SetParametersAsync(parameters);
        }

        protected override Task OnAfterRenderAsync(bool firstRender)
        {
            return base.OnAfterRenderAsync(firstRender);
        }
    }
}
```

```

    }

    protected override bool ShouldRender()
    {
        return base.ShouldRender();
    }
}

```

Let's analyze what the **CheckboxChanged** method does. First, the current value of the check box is assigned to the **n** variable.

```
var n = (bool)e.Value
```

Then, the value of the **n** variable is assigned to **Forecast.ShowExtras**. This is what hides the extras section for that row.

So, to see this in action, run the application with **Ctrl+F5** and navigate to the **Weather forecast** page. Then, click on any of the arrow buttons in the data table.

Date	Temp. (C)	Temp. (F)	Summary	
20 Oct 2020	23	73	Hot	↓
Humidity: 18%	Wind: Bursts	<input checked="" type="checkbox"/>		

Figure 4-e: Extras Section Shown

After the data row is displayed, click on the check box. As soon as you do that, the extras section for that row will be hidden.

Awesome—this solution works. However, we have not implemented two-way binding yet, because we had to add the **CheckboxChanged** method to be able to change the value of the **Forecast.ShowExtras** property.

Implementing two-way data binding happens when the value of the check box automatically changes the value of the **Forecast.ShowExtras** property, without the need to use a method.

So, how can we achieve this with Blazor? We need to use the **bind** directive. Let's update the **ForecastRow.razor** code to do that. Let's replace the following code:

```
<input type="checkbox" checked="@Forecast.ShowExtras"
      @onchange="e => base.CheckboxChanged(e)"/>
```

With this:

```
<input type="checkbox" @bind="Forecast.ShowExtras" />
```


So, what happened to the **onchange** event? Well, that's the beauty of two-way data binding—we don't need to invoke the **CheckboxChanged** method anymore.

We could even remove the **CheckboxChanged** method from the **ForecastRowBase** class, but let's leave it.

If you run the application by pressing **Ctrl+F5**, navigate to the **Weather forecast** page, click on any of the arrows, and then click on the check box, you'll see that it works as expected. That's the magic of two-way data binding.

Event callbacks

On some occasions, parent components need to be notified of events that happen in a child component so that the parent component can execute a specific action. This is where event callbacks come in handy.

For example, let's add a feature to our application that notifies the parent component, **FetchData**, and executes an action, such as displaying a notification or dialog, whenever the user clicks on the check box that is part of the **ForecastRow** component.

Of course, for this specific example, we could build the notification mechanism within the **ForecastRow** component. However, the idea is to showcase how a child component can notify a parent component that an event has taken place, so the parent component can execute an action.

Let's go ahead and implement this. Using **Solution Explorer**, go to the **Components** folder and open the **ForecastRow.razor.cs** file.

Replace the existing code with the following. The changes are highlighted in bold.

Code Listing 4-q: Updated ForecastRow.razor.cs

```
using BlazorApp.Data;
using Microsoft.AspNetCore.Components;
using System.Threading.Tasks;

namespace BlazorApp.Components
{
    public class ForecastRowBase: ComponentBase
    {
        [Parameter]
        public WeatherForecast Forecast { get; set; }

        protected WeatherExtras Extras { get; set; }

        [Inject]
        public WeatherExtrasService ExtrasService { get; set; }
```

```

[Parameter]
public EventCallback<bool> OnShowHide { get; set; }

protected async override Task OnInitializedAsync()
{
    Extras = await ExtrasService.GetExtrasAsync(Forecast);
}

public void CheckboxChanged(ChangeEventArgs e)
{
    var n = (bool)e.Value;
    Forecast.ShowExtras = n;
}

public void CheckboxClicked()
{
    OnShowHide.InvokeAsync(Forecast.ShowExtras);
}

protected override Task OnParametersSetAsync()
{
    return base.OnParametersSetAsync();
}

public override Task SetParametersAsync(ParameterView parameters)
{
    return base.SetParametersAsync(parameters);
}

protected override Task OnAfterRenderAsync(bool firstRender)
{
    return base.OnAfterRenderAsync(firstRender);
}

protected override bool ShouldRender()
{
    return base.ShouldRender();
}
}

```

If we carefully look at the preceding code, we can see that there are two new additions to the code.

The first one to notice is the **EventCallback** property called **OnShowHide**, which is annotated with the **Parameter** attribute.

```
[Parameter]
public EventCallback<bool> OnShowHide { get; set; }
```

As you know, the **Parameter** attribute indicates that the property can be passed from the parent to the child component.

The child component is where the property resides, which in this case is the **ForecastRow** component. The property is declared within the **ForecastRowBase** class, which **ForecastRow** inherits from.

The **EventCallback** is a delegate that indicates that the parent component (in our case the **FetchData** component) must implement an event to trigger the execution of a method when the child component changes the value of a property.

So, in other words, when the value of the **OnShowHide** property changes in the **ForecastRow** component, the parent component **FetchData** gets notified of this change and executes a method to perform some action.

Now, you might be asking yourself, what makes the value of the **OnShowHide** property change? That's the responsibility of the **CheckboxClicked** method.

```
public void CheckboxClicked()
{
    OnShowHide.InvokeAsync(Forecast.ShowExtras);
}
```

The **CheckboxClicked** method invokes the **OnShowHide.InvokeAsync** method, passing the value of **Forecast.ShowExtras** as a parameter. This is possible because **OnShowHide** is a delegate of type **EventCallback**.

The following diagram illustrates how this sequence of events happens between the **ForecastRow** and **FetchData** components.

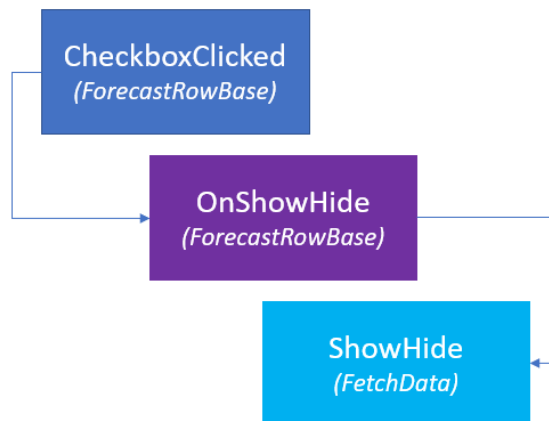


Figure 4-f: EventCallback Sequence (ForecastRowBase–FetchData)

But another question arises. What triggers the execution of the **CheckboxClicked** method? For that to happen, we need to bind the **CheckboxClicked** method to the input check box component.

To do that, using **Solution Explorer**, navigate to the **Components** folder and double-click on the **ForecastRow.razor** file to open it.

Replace the existing code with the following. The change is highlighted in bold.

Code Listing 4-r: Updated ForecastRow.razor

```
@inherits ForecastRowBase

<tr>
    <td>@Forecast.Date.ToShortDateString()</td>
    <td>@Forecast.TemperatureC</td>
    <td>@Forecast.TemperatureF</td>
    <td>@Forecast.Summary</td>
    <div @onclick="e => Forecast.ShowExtras = !Forecast.ShowExtras">
        <IconButton aria-label="delete"
            Style="margin: 8px;"
            Size="@IconButtonSize.Small">
            <ArrowDownwardIcon Style="font-size: inherit;" />
        </IconButton>
    </div>
</tr>
@if (Forecast.ShowExtras)
{
    <tr>
        <td>
            <strong>Humidity:</strong> @Extras.Humidity%
        </td>
        <td>
            <strong>Wind:</strong> @Extras.Wind
        </td>
        <td>
            <input type="checkbox" @bind="Forecast.ShowExtras"
                @onclick="base.CheckboxClicked" />
        </td>
    </tr>
}
```

Notice that we now have the **CheckboxClicked** method bound to the **onclick** event of the check box component.

This means that when the check box component is clicked, the **CheckboxClicked** method will initiate an event that is going to be received by the parent component, **FetchData**, which then executes a method. Let's have a look at it.

So, using **Solution Explorer**, open the **FetchData.razor** file that resides in the **Pages** folder and replace the existing code with the code from the following listing. The changes are highlighted in bold.

Code Listing 4-s: Updated FetchData.razor

```
@page "/fetchdata"

@using BlazorApp.Data
@inject WeatherForecastService ForecastService

@using Microsoft.Extensions.Logging;
@inject ILogger<Counter> logger;

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <ForecastRow @key="forecast" Forecast="forecast"
                    OnShowHide="ShowHide" />
            }
        </tbody>
    </table>
}
```

```

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }

    protected void ShowHide(bool toggle)
    {
        logger.LogWarning("Event callback triggered...");
    }
}

```

Let's have a look at what we have done. First, we imported the **Logging** namespace and injected the **logger** service into this component by invoking the following instructions.

```

@using Microsoft.Extensions.Logging;
@Inject ILogger<Counter> logger;

```

The only reason for doing this is that when the event is triggered, we want to log a message to the output console.

Next, we added the **OnShowHide** parameter to the **ForecastRow** component and assigned it to the **ShowHide** method.

```

<ForecastRow @key="forecast" Forecast="forecast" OnShowHide="ShowHide" />

```

After that, we implemented the **ShowHide** method in the **@code** section of the **FetchData** page component.

Notice that the **ShowHide** method receives as a parameter the Boolean value (**toggle**) that was passed to the **OnShowHide.InvokeAsync** method, which is the value of **Forecast.ShowExtras**.

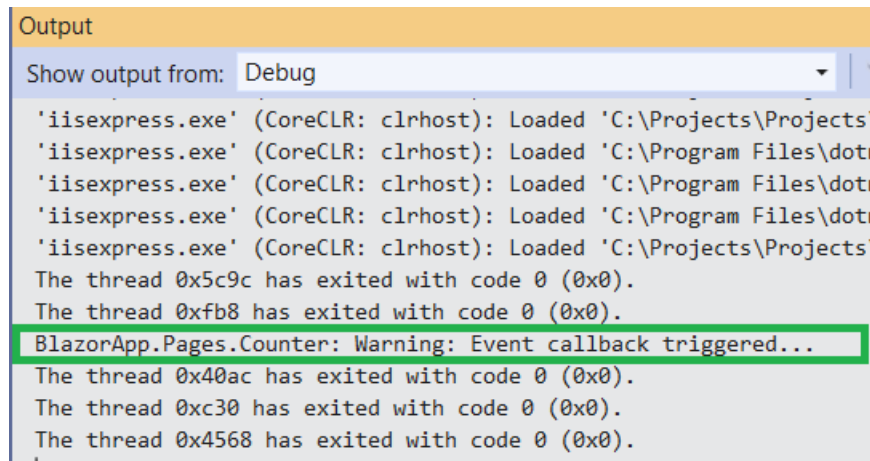
All this method does is write a message to the output console when **OnShowHide** is triggered.

Because we are just using this method to write to the console, we don't need to use the value of the **toggle** parameter. In any case, it is good to have it for future use.

For Blazor Server applications, the **logger** object writes messages to the output console of Visual Studio, whereas for Blazor WebAssembly applications, the message would appear on the browser's developer tools console.

So, to see this in action, let's run the application. This time, let's execute it in full debug mode by pressing **F5** and not **Ctrl+F5**, so we can see the message on the output console in Visual Studio when the event is triggered.

Once the application is running, navigate to the **Weather forecast** page and click on one of the arrow buttons. Once the extras section is displayed, click on the check box. You should then see the message written to the output console.



```
Output
Show output from: Debug
'iisexpress.exe' (CoreCLR: clrhost): Loaded 'C:\Projects\Projects\
'iisexpress.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotn
'iisexpress.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotn
'iisexpress.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotn
'iisexpress.exe' (CoreCLR: clrhost): Loaded 'C:\Projects\Projects\
The thread 0x5c9c has exited with code 0 (0x0).
The thread 0xfb8 has exited with code 0 (0x0).
BlazorApp.Pages.Counter: Warning: Event callback triggered...
The thread 0x40ac has exited with code 0 (0x0).
The thread 0xc30 has exited with code 0 (0x0).
The thread 0x4568 has exited with code 0 (0x0).
```

Figure 4-g: The Event Callback Message

Awesome, we have now seen how event callbacks can be used in Blazor to execute an action on a parent component that has been initiated from a change within a child component.

Combining parameters

So far in all the examples we have written, we've passed to our custom components, such as **ForecastRow**, a couple of parameters max.

However, in a real-life application, we could encounter a situation where a child component might require many parameters to be passed from the parent component. The more parameters to pass, the trickier it becomes to manage them.

Let's take this hypothetical situation, for instance, where we need to call the **ForecastRow** component from the parent **FetchData** component, as follows.

```
<ForecastRow @key="forecast" Forecast="forecast" Region="reg" City="city"
             State="state" Day="day" Month="month" Year="year" OnShowHide="ShowHide" />
```

This means that the **ForecastRowBase** class would look as follows.

```
[Parameter]
public string Region { get; set; }
```

```
[Parameter]
public string City { get; set; }
```

```
[Parameter]
public string State { get; set; }
```

```
[Parameter]
public string Day { get; set; }
```

```
[Parameter]
public string Month { get; set; }
```

```
[Parameter]
public string Year { get; set; }
```

```
[Parameter]
public WeatherForecast Forecast { get; set; }
```

What happens if the application requirements change and we need to add more parameters to the **ForecastRow** component?

You can start to see where this might be going, and the code may start to become messy and hard to manage fairly quickly.

Luckily, Blazor has a feature that can allow us to combine multiple parameters into one, making it much easier to manage how we can invoke a component like this one.

The way to do this is by using a **Dictionary** that combines all parameters into one, which would be as follows.

```
[Parameter]
public Dictionary<string, object> InputAttributes { get; set; } =
    new Dictionary<string, object>()
    {
        { "Region", "" },
        { "City", "" },
        { "State", "" },
        { "Day", "" },
        { "Month", "" },
        { "Year", "" }
    };
```

The key of the **Dictionary** is of type **string** and the value of type **object**. The names of the parameters become the keys within the **Dictionary**, and if we wanted to, we could give those keys default values, which in this case are empty.

So, once we have this **Dictionary**, we can invoke the child component by removing all the other parameters and replacing them with **@attributes="InputAttributes"**. Let's have a look.

```
<ForecastRow @key="forecast" Forecast="forecast"
    InputAttributes="inputAttributes" OnShowHide="ShowHide" />
```


Although we won't be doing anything with these extra parameters, let's update our code so we can have them ready just in case they would be required in the future.

So, let's open the **ForecastRow.razor.cs** file that resides in the **Components** folder and replace the existing code with the following. The changes are highlighted in bold.

Code Listing 4-t: Updated ForecastRow.razor.cs

```
using BlazorApp.Data;
using Microsoft.AspNetCore.Components;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace BlazorApp.Components
{
    public class ForecastRowBase: ComponentBase
    {
        [Parameter]
        public Dictionary<string, object> InputAttributes { get; set; } =
            new Dictionary<string, object>()
            {
                { "Region", "" },
                { "City", "" },
                { "State", "" },
                { "Day", "" },
                { "Month", "" },
                { "Year", "" }
            };

        [Parameter]
        public WeatherForecast Forecast { get; set; }

        protected WeatherExtras Extras { get; set; }

        [Inject]
        public WeatherExtrasService ExtrasService { get; set; }

        [Parameter]
        public EventCallback<bool> OnShowHide { get; set; }

        protected async override Task OnInitializedAsync()
        {
            Extras = await ExtrasService.GetExtrasAsync(Forecast);
        }
    }
}
```

```

    public void CheckboxChanged(ChangeEventArgs e)
    {
        var n = (bool)e.Value;
        Forecast.ShowExtras = n;
    }

    public void CheckboxClicked()
    {
        OnShowHide.InvokeAsync(Forecast.ShowExtras);
    }

    protected override Task OnParametersSetAsync()
    {
        return base.OnParametersSetAsync();
    }

    public override Task SetParametersAsync(ParameterView parameters)
    {
        return base.SetParametersAsync(parameters);
    }

    protected override Task OnAfterRenderAsync(bool firstRender)
    {
        return base.OnAfterRenderAsync(firstRender);
    }

    protected override bool ShouldRender()
    {
        return base.ShouldRender();
    }
}

```

Notice that the only changes we have made are adding the reference to the **Generic** namespace, declaring the **InputAttributes** as a **Dictionary**, and decorating it with the **Parameter** attribute.

Then, on the parent component **FetchData**, the code also needs to be replaced and updated as follows. The changes are highlighted in bold.

Code Listing 4-u: Updated FetchData.razor.cs

```
@page "/fetchdata"
```

```

@using BlazorApp.Data
@Inject WeatherForecastService ForecastService

@using Microsoft.Extensions.Logging;
@Inject ILogger<Counter> logger;

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <ForecastRow @key="forecast" Forecast="forecast"
                    InputAttributes="inputAttributes"
                    OnShowHide="ShowHide" />
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[] forecasts;
    private Dictionary<string, object> inputAttributes;

    protected override async Task OnInitializedAsync()
    {
        inputAttributes = new Dictionary<string, object>()
        {

```

```

        { "Region", "Spain" },
        { "City", "Valencia" },
        { "State", "Comunidad Valenciana" },
        { "Day", "01" },
        { "Month", "11" },
        { "Year", "2020" }
    };

    forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
}

protected void ShowHide(bool toggle)
{
    logger.LogWarning("Event callback triggered...");
}
}

```

So, by looking at this code, we can see that the first change has to do with assigning the value of the `inputAttributes` (which is declared in the `@code` section of the component) to the child component, `ForecastRow`.

```

<ForecastRow @key="forecast" Forecast="forecast"
    InputAttributes="inputAttributes" OnShowHide="ShowHide" />

```

Next, `inputAttributes` is declared as a `Dictionary` in the `@code` section.

```
private Dictionary<string, object> inputAttributes;
```

Following that, within the `OnInitializedAsync` method, `inputAttributes` is initialized with default values.

If we now run the application using **Ctrl+F5**, everything should continue to work as expected. There won't be any visible changes to the UI of the application, though.

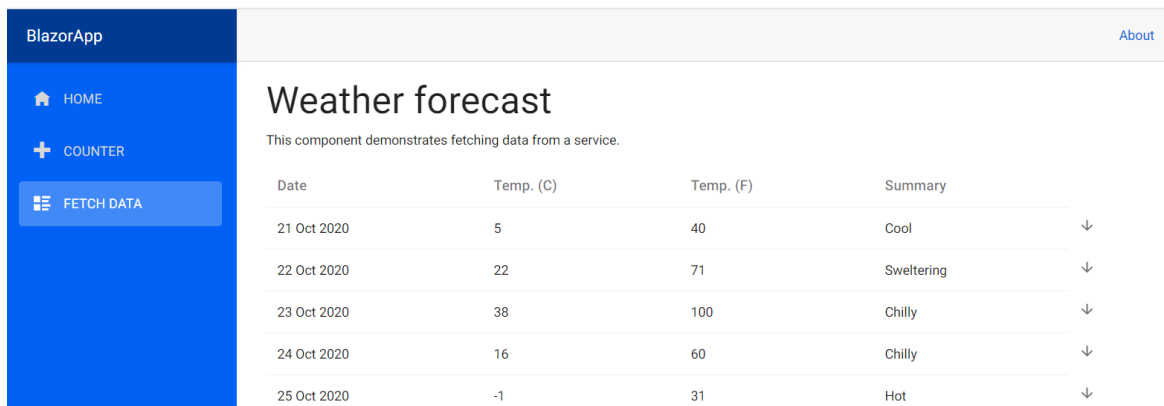


Figure 4-h: The App Running as Expected (Weather forecast)

So, by using this technique, Blazor offers us a way to easily handle and effectively manage child components that have many parameters.

Referencing components

There's one last technique I'd like to talk about that can also come in handy when working with real-life projects. That is to get a reference to a child component that can be used to access members of it within a parent component.

Let's say that we want to create a new component called **CoolDialog** to display a fancy Material Design dialog box, which is going to be our child component.

Then we want to be able to invoke one of the methods within the **CoolDialog** child component that will display the dialog, but we want to invoke that method from the parent component—in this case from the **FetchData** page component. So, how can we do this?

Using **Solution Explorer**, click on the **Components** folder, right-click, click **Add**, then click **Razor Component**. Let's call the file **CoolDialog.razor** and then click **Add**.

Once the file has been created, let's add the following code to it.

Code Listing 4-v: CoolDialog.razor

```
@inherits CoolDialogBase

<Skclusive.Material.Dialog.Dialog Open="@base.Open"
    OnClose="@base.OnClose"
    aria-labelledby="alert-dialog-title"
    aria-describedby="alert-dialog-description">
    <DialogTitle id="alert-dialog-title">
        Do you like this dialog?
    </DialogTitle>
    <DialogContent>
        <DialogContentText id="alert-dialog-description">
            This is a dialog from the Skclusive-UI library :)
        </DialogContentText>
    </DialogContent>
    <DialogActions>
        <Button OnClick="@base.OnClose"
            Color="@Color.Primary">
            Disagree
        </Button>
        <Button OnClick="@base.OnClose"
            Color="@Color.Primary"
            autofocus>
            Agree
    </DialogActions>
</Skclusive.Material.Dialog.Dialog>
```

```
</Button>
</DialogActions>
</Skclusive.Material.Dialog.Dialog>
```

The preceding code simply uses components from the Skclusive-UI library to create the HTML markup for a dialog box.

From this code, two things stand out right away. One is that the code references a property called **base.Open**, and the other is that there's a reference to a **base.OnClose** method.

Both refer to the **base** class of this component, which we haven't created yet and is going to be called **CoolDialogBase**, which the component inherits from (**@inherits CoolDialogBase**). So, let's create the **base** class and code-behind file for this component.

Using **Solution Explorer**, click on the **Components** folder, right-click, click **Add**, then click **New Item**. From the list of items, choose **Class**.

Let's call the file **CoolDialog.razor.cs** and then click **Add**. Once the file has been created, open it and add the following code.

Code Listing 4-w: CoolDialog.razor.cs

```
using Microsoft.AspNetCore.Components;

namespace BlazorApp.Components
{
    public class CoolDialogBase: ComponentBase
    {
        protected bool Open { set; get; }

        public void OnClose()
        {
            Open = false;

            StateHasChanged();
        }

        public void OnOpen()
        {
            Open = true;

            StateHasChanged();
        }
    }
}
```

Following the same naming convention we've used all along with this book, we'll add the suffix **Base** to the name of the class. So, in this case, the class is called **CoolDialogBase**.

The code is very simple, and there are two methods for opening (**OnOpen**) and closing (**OnClose**) the dialog box, as well as a Boolean property (**Open**) that indicates whether the dialog is open or closed.

When the opening (**OnOpen**) and closing (**OnClose**) methods are executed, the component's state is changed, which is done by invoking the **StateHasChanged** method.

Great—we now have a dialog box we can use within our application. So, to use it, go to **Solution Explorer** and under the **Pages** folder, open the **FetchData.razor** file.

Once the file is open, replace the existing code with the following. The changes have been highlighted in bold.

Code Listing 4-x: Updated FetchData.razor

```
@page "/fetchdata"

@using BlazorApp.Data
@inject WeatherForecastService ForecastService

@using Microsoft.Extensions.Logging;
@inject ILogger<Counter> logger;

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
```

```

        {
            <ForecastRow @key="forecast" Forecast="forecast"
                InputAttributes="inputAttributes"
                OnShowHide="ShowHide" />
        }
    </tbody>
</table>
<CoolDialog @ref="coolDlg" />
}
@code {
    private WeatherForecast[] forecasts;
    private Dictionary<string, object> inputAttributes;
    private CoolDialogBase coolDlg;

    protected override async Task OnInitializedAsync()
    {
        inputAttributes = new Dictionary<string, object>()
        {
            { "Region", "Spain" },
            { "City", "Valencia" },
            { "State", "Comunidad Valenciana" },
            { "Day", "01" },
            { "Month", "11" },
            { "Year", "2020" }
        };

        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }

    protected void ShowHide(bool toggle)
    {
        logger.LogWarning("Event callback triggered...");
        coolDlg.OnOpen();
    }
}

```

The first change to the code that you'll notice is that the **CoolDialog** component has been added after the `<table/>` tag.

<CoolDialog @ref="coolDlg" />

The interesting thing about the way that this component has been added to the HTML markup is that a reference to an instance of the **CoolDialogBase** class is included in the markup.

This means that we can invoke accessible properties or methods of the **CoolDialogBase** class by using the **coolDlg** instance, which is declared within the **@code** section, as follows.

```
private CoolDialogBase coolDlg;
```

Although the component has been added to the HTML markup, it won't be visible by default when the application runs.

Finally, within **ShowHide**, the **OnOpen** method of the **CoolDialogBase** instance is invoked. So, that's it—we are ready to test the dialog box functionality using a child component reference from a parent component.

Let's run the application with **Ctrl+F5**, navigate to the **Weather forecast** page, click on one of the arrows, and then when the extras section is open, click on the check box.

Once the check box has been clicked, the dialog box will be displayed, which we can see in the following figure.

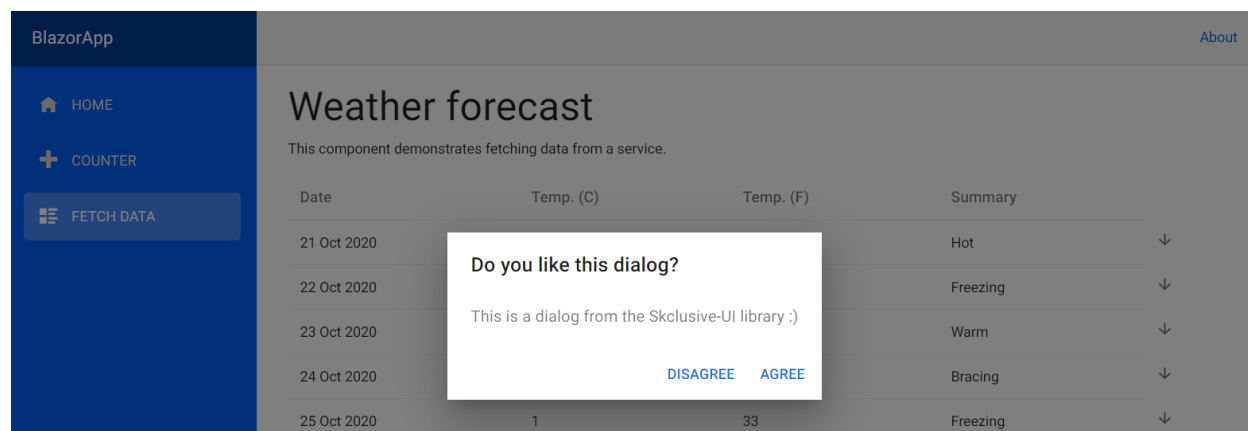


Figure 4-i: The CoolDialog Component (Invoked as a Reference from FetchData)

To close the dialog, you can click on any of the buttons. Awesome.

Summary

We've reached the end of this chapter. We explored many of the features and techniques in Blazor that allow us to compose an application by using various component techniques.

By having to compose components, we've seen the various ways data can pass from one component to another and how events can be cascaded from a child to a parent component, as well as how a parent component can invoke methods and properties contained within the child.

These techniques are essential for creating any Blazor application that should be as modular and reusable as possible using components.

In the next and final chapter, we will briefly explore the template syntax so you can create component templates.

Chapter 5 Templating

By using a feature of Blazor known as templates, it is easy to have a consistent UI throughout the various components of a Blazor application, which is what we are going to explore now.

In our case, our application has an HTML table with some data that is displayed on the **Weather forecast** page.

The table displays the information in a basic way that works fine, but it is not very appealing. The look and feel of the table could be improved.

The case for component templating

At the moment, our application has a single table, so we could add the CSS style and HTML markup improvements directly to the table and we would solve the problem, right?

In our app's current state, yes, that would be enough because the app has only one table. But what happens if we need to add additional tables to our application due to an increase in the scope or functionality of the app?

In such a scenario, we would need to apply CSS styling and improvements to the HTML markup on any additional tables as well. This means that in a way, we would be duplicating the effort required to bring all tables in our application to a common style or functionality.

This problem can be solved by using a feature that Blazor provides allowing us to create components based on a template.

So, effectively, instead of making changes when required to for every CSS style or HTML markup for each independent table, by using a component template, we only need to make these changes to the template, not every table. This can be a huge time-saver.

Table template

Creating a table component template is very easy. In the previous chapters, we already covered the fundamental techniques that allow us to achieve that. We just haven't seen how to apply them for templating purposes.

Using **Solution Explorer**, click on the **Shared** folder, right-click, then click **Add**, and then click **Razor Component**.

Let's call the new component **TableTemplate.razor**, and click **Add** to finish creating the file. Once the file has been created, open it and add the following code.

Code Listing 5-a: TableTemplate.razor

```
<table class="table">
  <thead>
    @TableHeader
  </thead>
  <tbody>
    @TableBody
  </tbody>
</table>

@code {
  [Parameter]
  public RenderFragment TableHeader { get; set; }

  [Parameter]
  public RenderFragment TableBody { get; set; }
}
```

As you can see, what we've done is taken the table HTML markup from the **FetchData.razor** file, removed the content from the **thead** and **tbody** sections, and added it as **RenderFragment** parameters, which will be passed from the parent component.

By making this simple change, we now have a table component template that we can use on any page of the application where we need to display a table.

Now, let's adjust the code within the **FetchData.razor** file to start using the table component template.

Using **Solution Explorer**, open the **FetchData.razor** file that resides in the **Pages** folder. Then, replace the existing code with the following. The changes are highlighted in bold.

Code Listing 5-b: Updated FetchData.razor

```
@page "/fetchdata"

@using BlazorApp.Data
@inject WeatherForecastService ForecastService

@using Microsoft.Extensions.Logging;
@inject ILogger<Counter> logger;

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>
```

```

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <TableTemplate>
        <TableHeader>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </TableHeader>
        <TableBody>
            @foreach (var forecast in forecasts)
            {
                <ForecastRow @key="forecast" Forecast="forecast"
                    InputAttributes="inputAttributes"
                    OnShowHide="ShowHide" />
            }
        </TableBody>
    </TableTemplate>
    <CoolDialog @ref="coolDlg" />
}

@code {
    private WeatherForecast[] forecasts;
    private Dictionary<string, object> inputAttributes;
    private CoolDialogBase coolDlg;

    protected override async Task OnInitializedAsync()
    {
        inputAttributes = new Dictionary<string, object>()
        {
            { "Region", "Spain" },
            { "City", "Valencia" },
            { "State", "Comunidad Valenciana" },
            { "Day", "01" },
            { "Month", "11" },
            { "Year", "2020" }
        };
    }
}

```

```

        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }

    protected void ShowHide(bool toggle)
    {
        logger.LogWarning("Event callback triggered...");
        coolDlg.OnOpen();
    }
}

```

So, what have we done here? Maybe the easiest way to understand the changes made is by looking at the following diagram.

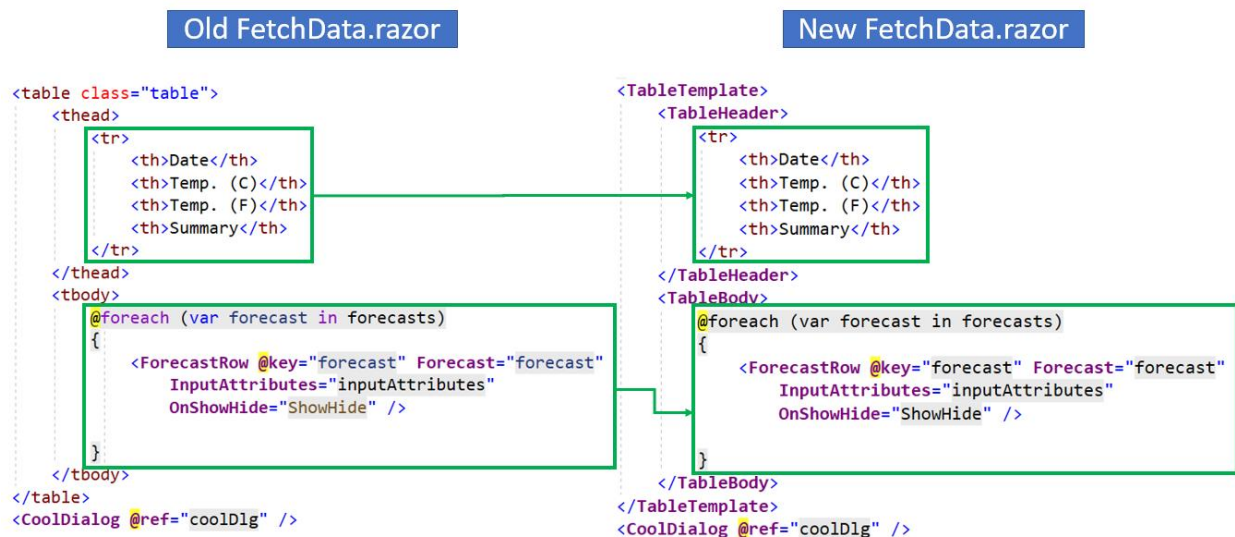


Figure 5-a: Old and New FetchData.razor Comparison

By looking at the diagram, we see that the amount of code between both approaches is almost the same.

So, you might be asking yourself, what was the point of doing all this if we ended up with the same amount of code on the **FetchData.razor** page, and on top of that, we had to create new files to host the **TableTemplate** component code?

Well, we have added a bit of extra code by creating the **TableTemplate** component, but if we have to create several additional tables for our application in the future, we'll end up saving a lot of time (and code), given that we can reuse the **TableTemplate** component for all other tables.

By using the **RenderFragment** properties, we can pass from the parent component (**FetchData**) to the child component (**TableTemplate**), the header section (by using **TableHeader**), and the body section (by using **TableBody**).

If we now run the application by pressing **Ctrl+F5**, and then navigate to the **Weather forecast** page, we can see that the application behaves the same way.

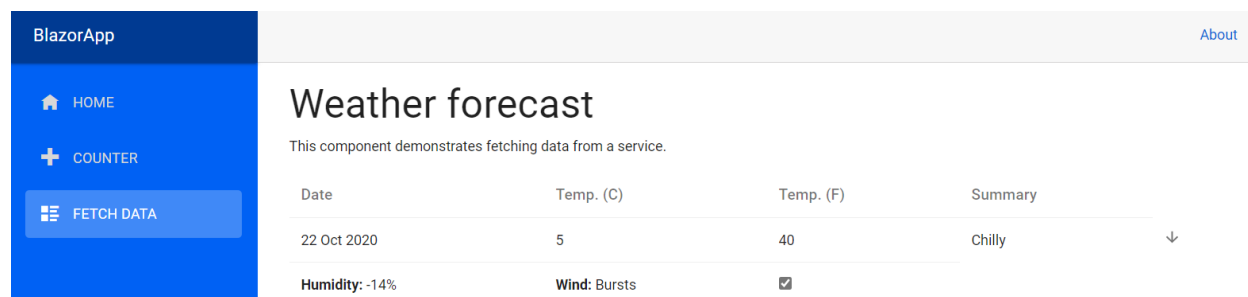


Figure 5-b: The FetchData Page Component Rendering the Table with the TableTemplate Component

Awesome—our application continues to work with its familiar look and feel.

Closing comments

We've reached the end of this brief but important chapter and also reached the end of this book. The book's goal was to highlight the benefits of working with components in Blazor applications using the Razor syntax—in other words, Razor components.

Throughout the book, we explored interesting techniques on how to structure parent and child components, and how to pass information between them, and explored various ways components can enhance Blazor applications by providing reusability, modularity, and better code organization.

Blazor is a fascinating technology that Microsoft has been working on for a couple of years. It has quickly evolved into a rapid application framework that has shortened the gap between back-end and front-end development.

Two excellent resources to help you stay up to date with the latest Blazor trends are the main [Blazor website](#) and the [Awesome Blazor GitHub](#) repo, which contains a wealth of links to other useful Blazor resources.

Although this book focused on a server-side Blazor app, the concepts covered here are mostly applicable when developing Blazor WebAssembly applications, as well. If you would like to learn more about Blazor WebAssembly development, the *Succinctly* series has you covered with [Blazor WebAssembly Succinctly](#) by Michael Washington, which gives you in-depth information on Blazor WebAssembly development in an easy-to-comprehend way.

I invite you to continue exploring the fascinating world of Blazor app development, and if you build something cool with the concepts covered in this book, please feel free to reach out to me at hello@edfreitas.me. I'd love to hear from you!

Thanks for reading this book. I hope you have enjoyed it, and until next time, go, be awesome.